目　录

简介

简介

beego 是免费、开源的软件，这意味着任何人都可以为其开发和进步贡献力量。beego 源代码目前托管在 Github 上，Github 提供非常容易的途径 fork 项目和合并你的贡献。

beego 安装

beego 安装升级

bee工具的使用

beego 安装升级

beego 安装升级

beego 的安装

beego 的安装是典型的 Go 安装包的形式：

```
go get github.com/astaxie/beego
```

常见问题：

- git 没有安装，请自行安装不同平台的 git，如何安装请自行搜索。
- git https 无法获取，请配置本地的 git，关闭 https 验证：

  ```
  git config --global http.sslVerify false
  ```

- 无法上网怎么安装 beego，目前没有好的办法，接下来我们会整理一个全包下载，每次发布正式版本都会提供这个全包下载，包含依赖包。

beego 的升级

beego 升级分为 go 方式升级和源码下载升级：

- Go 升级,通过该方式用户可以升级 beego 框架，强烈推荐该方式：

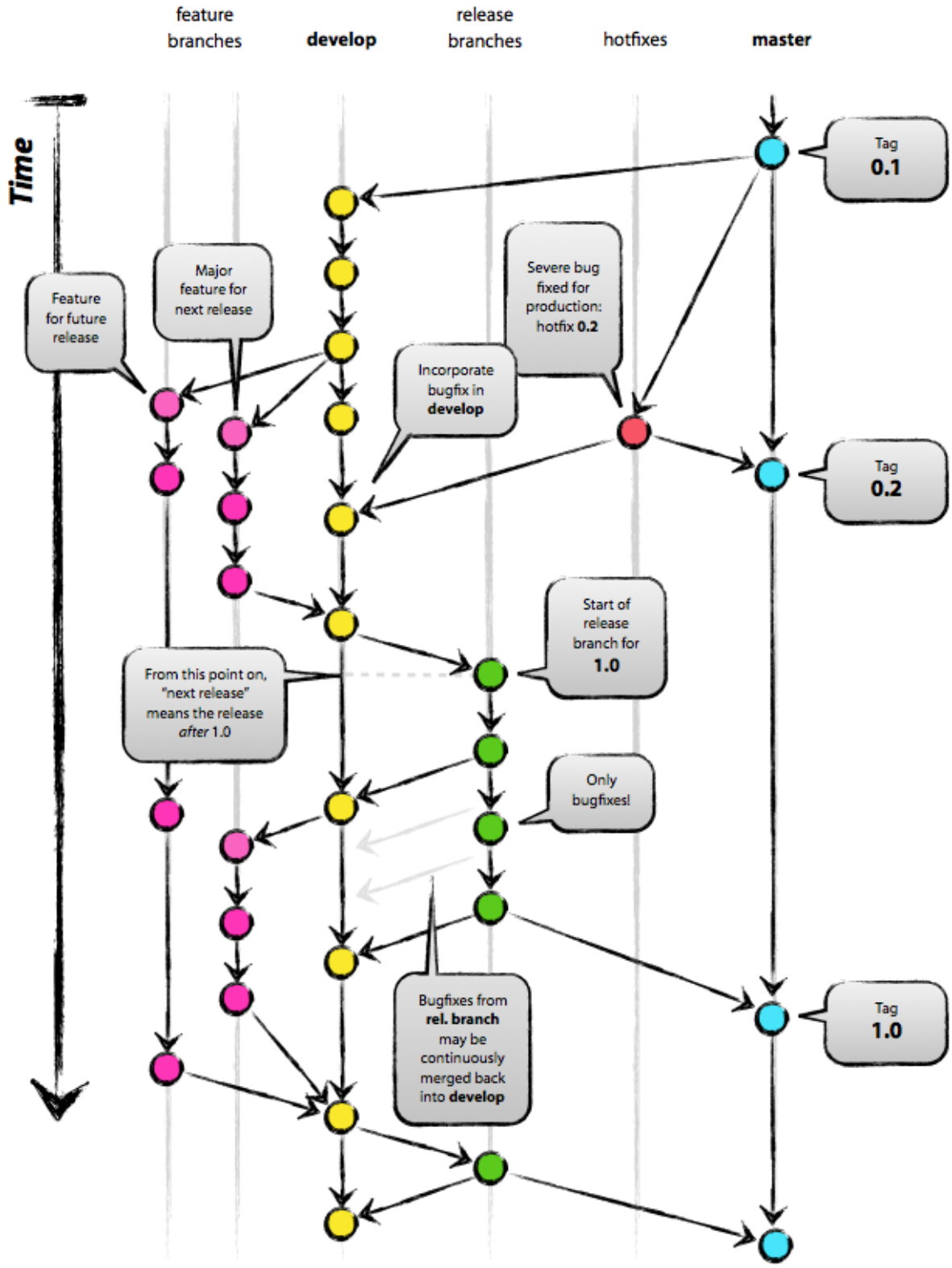
  ```
  go get -u github.com/astaxie/beego
  ```

- 源码下载升级，用户访问 `https://github.com/astaxie/beego` ,下载源码，然后覆盖到 `$GOPATH/src/github.com/astaxie/beego` 目录，然后通过本地执行安装就可以升级了：

  ```
  go install     github.com/astaxie/beego
  ```

beego 的 git 分支

beego 的 master 分支为相对稳定版本，dev 分支为开发者版本。大致流程如下：



bee工具的使用

bee工具的使用

bee 工具简介

bee 工具是一个为了协助快速开发 beego 项目而创建的项目，通过 bee 您可以很容易的进行 beego 项目的创建、热编译、开发、测试、和部署。

bee 工具的安装

您可以通过如下的方式安装 bee 工具：

```
go get github.com/beego/bee
```

安装完之后，`bee` 可执行文件默认存放在 `$GOPATH/bin` 里面，所以您需要把 `$GOPATH/bin` 添加到您的环境变量中，才可以进行下一步。

> s="default">
>
> s="default">
>
> 如何添加环境变量，请自行搜索
> 如果你本机设置了 `GOBIN` ，那么上面的命令就会安装到 `GOBIN` 下，请添加 GOBIN 到你的环境变量中

bee 工具命令详解

我们在命令行输入 `bee` ，可以看到如下的信息：

```
Bee is a Fast and Flexible tool for managing your Beego Web Application.

Usage:

    bee command [arguments]

The commands are:
```

```
    version     show the bee & beego version
    migrate     run database migrations
    api         create an api application base on beego framework
    bale        packs non-Go files to Go source files
    new         create an application base on beego framework
    run         run the app which can hot compile
    pack        compress an beego project
    fix         Fixes your application by making it compatible with newer versio
ns of Beego
    dlv         Start a debugging session using Delve
    dockerize   Generates a Dockerfile for your Beego application
    generate    Source code generator
    hprose      Creates an RPC application based on Hprose and Beego frameworks
    new         Creates a Beego application
    pack        Compresses a Beego application into a single file
    rs          Run customized scripts
    run         Run the application by starting a local development server
    server      serving static content over HTTP on port

Use bee help [command] for more information about a command.
```

`new` 命令

`new` 命令是新建一个 Web 项目，我们在命令行下执行 `bee new <项目名>` 就可以创建一个新的项目。但是注意该命令必须在 `$GOPATH/src` 下执行。最后会在 `$GOPATH/src` 相应目录下生成如下目录结构的项目：

```
bee new myproject
[INFO] Creating application...
/gopath/src/myproject/
/gopath/src/myproject/conf/
/gopath/src/myproject/controllers/
/gopath/src/myproject/models/
/gopath/src/myproject/static/
/gopath/src/myproject/static/js/
/gopath/src/myproject/static/css/
/gopath/src/myproject/static/img/
/gopath/src/myproject/views/
/gopath/src/myproject/conf/app.conf
/gopath/src/myproject/controllers/default.go
/gopath/src/myproject/views/index.tpl
/gopath/src/myproject/main.go
13-11-25 09:50:39 [SUCC] New application successfully created!
```

```
myproject
├──── conf
|     └──── app.conf
├──── controllers
|     └──── default.go
├──── main.go
├──── models
├──── routers
|     └──── router.go
├──── static
|     ├──── css
|     ├──── img
|     └──── js
├──── tests
|     └──── default_test.go
└──── views
      └──── index.tpl

8 directories, 4 files
```

`api` 命令

上面的 `new` 命令是用来新建 Web 项目，不过很多用户使用 beego 来开发 API 应用。所以这个 `api` 命令就是用来创建 API 应用的，执行命令之后如下所示：

```
bee api apiproject
create app folder: /gopath/src/apiproject
create conf: /gopath/src/apiproject/conf
create controllers: /gopath/src/apiproject/controllers
create models: /gopath/src/apiproject/models
create tests: /gopath/src/apiproject/tests
create conf app.conf: /gopath/src/apiproject/conf/app.conf
create controllers default.go: /gopath/src/apiproject/controllers/default.go
create tests default.go: /gopath/src/apiproject/tests/default_test.go
create models object.go: /gopath/src/apiproject/models/object.go
create main.go: /gopath/src/apiproject/main.go
```

这个项目的目录结构如下：

```
apiproject
├──── conf
|     └──── app.conf
├──── controllers
|     └──── object.go
```

```
|   └── user.go
├── docs
|   └── doc.go
├── main.go
├── models
|   └── object.go
|   └── user.go
├── routers
|   └── router.go
└── tests
    └── default_test.go
```

从上面的目录我们可以看到和 Web 项目相比，少了 static 和 views 目录，多了一个 test 模块，用来做单元测试的。

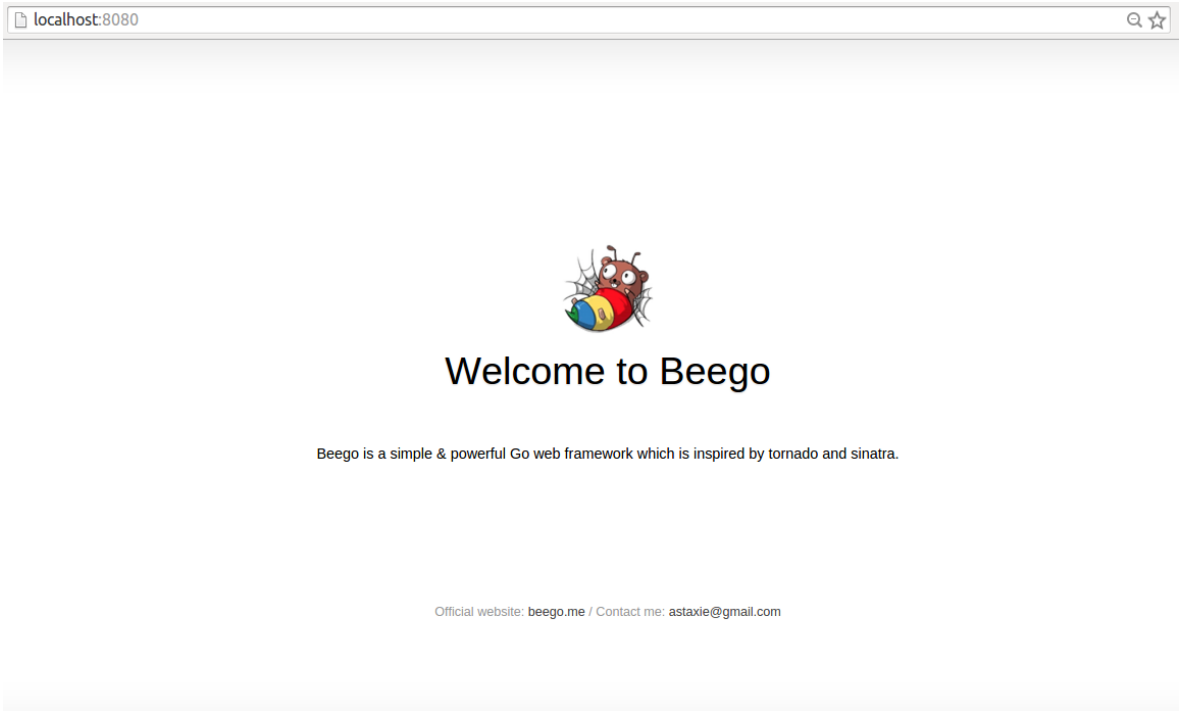
同时，该命令还支持一些自定义参数自动连接数据库创建相关 model 和 controller:
`bee api [appname] [-tables=""] [-driver=mysql] [-conn="root:<password>@tcp(127.0.0.1:3306)/test"]`
如果 conn 参数为空则创建一个示例项目，否则将基于链接信息链接数据库创建项目。

`run` 命令

我们在开发 Go 项目的时候最大的问题是经常需要自己手动去编译再运行，`bee run` 命令是监控 beego 的项目，通过 fsnotify监控文件系统。但是注意该命令必须在 `$GOPATH/src/appname` 下执行。
这样我们在开发过程中就可以实时的看到项目修改之后的效果：

```
bee run
13-11-25 09:53:04 [INFO] Uses 'myproject' as 'appname'
13-11-25 09:53:04 [INFO] Initializing watcher...
13-11-25 09:53:04 [TRAC] Directory(/gopath/src/myproject/controllers)
13-11-25 09:53:04 [TRAC] Directory(/gopath/src/myproject/models)
13-11-25 09:53:04 [TRAC] Directory(/gopath/src/myproject)
13-11-25 09:53:04 [INFO] Start building...
13-11-25 09:53:16 [SUCC] Build was successful
13-11-25 09:53:16 [INFO] Restarting myproject ...
13-11-25 09:53:16 [INFO] ./myproject is running...
```

我们打开浏览器就可以看到效果 `http://localhost:8080/` :



如果我们修改了 `Controller` 下面的 `default.go` 文件，我们就可以看到命令行输出：

```
13-11-25 10:11:20 [EVEN] "/gopath/src/myproject/controllers/default.go": DELETE|
MODIFY
13-11-25 10:11:20 [INFO] Start building...
13-11-25 10:11:20 [SKIP] "/gopath/src/myproject/controllers/default.go": CREATE
13-11-25 10:11:23 [SKIP] "/gopath/src/myproject/controllers/default.go": MODIFY
13-11-25 10:11:23 [SUCC] Build was successful
13-11-25 10:11:23 [INFO] Restarting myproject ...
13-11-25 10:11:23 [INFO] ./myproject is running...
```

刷新浏览器我们看到新的修改内容已经输出。

`pack` 命令

`pack` 目录用来发布应用的时候打包，会把项目打包成 zip 包，这样我们部署的时候直接把打包之后的项目上传，解压就可以部署了：

```
bee pack
app path: /gopath/src/apiproject
GOOS darwin GOARCH amd64
build apiproject
build success
exclude prefix:
```

```
exclude suffix: .go:.DS_Store:.tmp
file write to `/gopath/src/apiproject/apiproject.tar.gz`
```

我们可以看到目录下有如下的压缩文件：

```
rwxr-xr-x  1 astaxie  staff  8995376 11 25 22:46 apiproject
-rw-r--r--  1 astaxie  staff  2240288 11 25 22:58 apiproject.tar.gz
drwxr-xr-x  3 astaxie  staff      102 11 25 22:31 conf
drwxr-xr-x  3 astaxie  staff      102 11 25 22:31 controllers
-rw-r--r--  1 astaxie  staff      509 11 25 22:31 main.go
drwxr-xr-x  3 astaxie  staff      102 11 25 22:31 models
drwxr-xr-x  3 astaxie  staff      102 11 25 22:31 tests
```

bale 命令

这个命令目前仅限内部使用，具体实现方案未完善，主要用来压缩所有的静态文件变成一个变量申明文件，全部编译到二进制文件里面，用户发布的时候携带静态文件，包括 js、css、img 和 views。最后在启动运行时进行非覆盖式的自解压。

version 命令

这个命令是动态获取 bee、beego 和 Go 的版本，这样一旦用户出现错误，可以通过该命令来查看当前的版本

```
$ bee version
bee    :1.2.2
beego  :1.4.2
Go     :go version go1.3.3 darwin/amd64
```

generate 命令

这个命令是用来自动化的生成代码的，包含了从数据库一键生成 model，还包含了 scaffold 的，通过这个命令，让大家开发代码不再慢

```
bee generate scaffold [scaffoldname] [-fields=""] [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    The generate scaffold command will do a number of things for you.
    -fields: a list of table fields. Format: field:type, ...
    -driver: [mysql | postgres | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp(127.0.0.1:3306)/test
    example: bee generate scaffold post -fields="title:string,body:text"
```

```
bee generate model [modelname] [-fields=""]
    generate RESTful model based on fields
    -fields: a list of table fields. Format: field:type, ...

bee generate controller [controllerfile]
    generate RESTful controllers

bee generate view [viewpath]
    generate CRUD view in viewpath

bee generate migration [migrationfile] [-fields=""]
    generate migration file for making database schema update
    -fields: a list of table fields. Format: field:type, ...

bee generate docs
    generate swagger doc file

bee generate test [routerfile]
    generate testcase

bee generate appcode [-tables=""] [-driver=mysql] [-conn="root:@tcp(127.0.0.1:33
06)/test"] [-level=3]
    generate appcode based on an existing database
    -tables: a list of table names separated by ',', default is empty, indicatin
g all tables
    -driver: [mysql | postgres | sqlite], the default is mysql
    -conn:   the connection string used by the driver.
             default for mysql:    root:@tcp(127.0.0.1:3306)/test
             default for postgres: postgres://postgres:postgres@127.0.0.1:5432/p
ostgres
    -level:  [1 | 2 | 3], 1 = models; 2 = models,controllers; 3 = models,control
lers,router
```

`migrate` 命令

这个命令是应用的数据库迁移命令，主要是用来每次应用升级，降级的SQL管理。

```
bee migrate [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    run all outstanding migrations
    -driver: [mysql | postgresql | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp
(127.0.0.1:3306)/test

bee migrate rollback [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    rollback the last migration operation
```

```
    -driver: [mysql | postgresql | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp(
127.0.0.1:3306)/test

bee migrate reset [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    rollback all migrations
    -driver: [mysql | postgresql | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp(
127.0.0.1:3306)/test

bee migrate refresh [-driver=mysql] [-conn="root:@tcp(127.0.0.1:3306)/test"]
    rollback all migrations and run them all again
    -driver: [mysql | postgresql | sqlite], the default is mysql
    -conn:   the connection string used by the driver, the default is root:@tcp(
127.0.0.1:3306)/test
```

`dockerize` 命令

这个命令可以通过生成Dockerfile文件来实现docker化你的应用。

例子:
生成一个以1.6.4版本Go环境为基础镜像的Dockerfile,并暴露9000端口:

```
$ bee dockerize -image="library/golang:1.6.4" -expose=9000
______
| ___ \
| |_/ /  ___   ___
| ___ \ / _ \ / _ \
| |_/ /|  __/|  __/
\____/  \___| \___| v1.6.2
2016/12/26 22:34:54 INFO     ▶ 0001 Generating Dockerfile...
2016/12/26 22:34:54 SUCCESS  ▶ 0002 Dockerfile generated.
```

更多帮助信息可执行 `bee help dockerize` .

bee 工具配置文件

您可能已经注意到，在 bee 工具的源码目录下有一个 `bee.json` 文件，这个文件是针对 bee 工具的一些行为进行配置。该功能还未完全开发完成，不过其中的一些选项已经可以使用：

- `"version": 0` ：配置文件版本，用于对比是否发生不兼容的配置格式版本。

- `"go_install": false`：如果您的包均使用完整的导入路径（例如：`github.com/user/repo/subpkg`）,则可以启用该选项来进行 `go install` 操作，加快构建操作。
- `"watch_ext": []`：用于监控其它类型的文件（默认只监控后缀为 `.go` 的文件）。
- `"dir_structure":{}`：如果您的目录名与默认的 MVC 架构的不同，则可以使用该选项进行修改。
- `"cmd_args": []`：如果您需要在每次启动时加入启动参数，则可以使用该选项。
- `"envs": []`：如果您需要在每次启动时设置临时环境变量参数，则可以使用该选项。

快速入门

新建项目

路由设置

Controller运行机制

Model逻辑

View编写

静态文件处理

新建项目

新建项目

创建项目

beego 的项目基本都是通过 `bee` 命令来创建的，所以在创建项目之前确保你已经安装了 bee 工具和 beego。如果你还没有安装，那么请查阅 beego 的安装 和 bee 工具的安装。

现在一切就绪我们就可以开始创建项目了，打开终端，进入 `$GOPATH/src` 所在的目录：

```
➜  src  bee new quickstart
[INFO] Creating application...
/gopath/src/quickstart/
/gopath/src/quickstart/conf/
/gopath/src/quickstart/controllers/
/gopath/src/quickstart/models/
/gopath/src/quickstart/routers/
/gopath/src/quickstart/tests/
/gopath/src/quickstart/static/
/gopath/src/quickstart/static/js/
/gopath/src/quickstart/static/css/
/gopath/src/quickstart/static/img/
/gopath/src/quickstart/views/
/gopath/src/quickstart/conf/app.conf
/gopath/src/quickstart/controllers/default.go
/gopath/src/quickstart/views/index.tpl
/gopath/src/quickstart/routers/router.go
/gopath/src/quickstart/tests/default_test.go
/gopath/src/quickstart/main.go
2014/11/06 18:17:09 [SUCC] New application successfully created!
```

通过一个简单的命令就创建了一个 beego 项目。他的目录结构如下所示

```
quickstart
|-- conf
|   `-- app.conf
|-- controllers
|   `-- default.go
|-- main.go
```

```
|-- models
|-- routers
|   `-- router.go
|-- static
|   |-- css
|   |-- img
|   `-- js
|-- tests
|   `-- default_test.go
`-- views
    `-- index.tpl
```

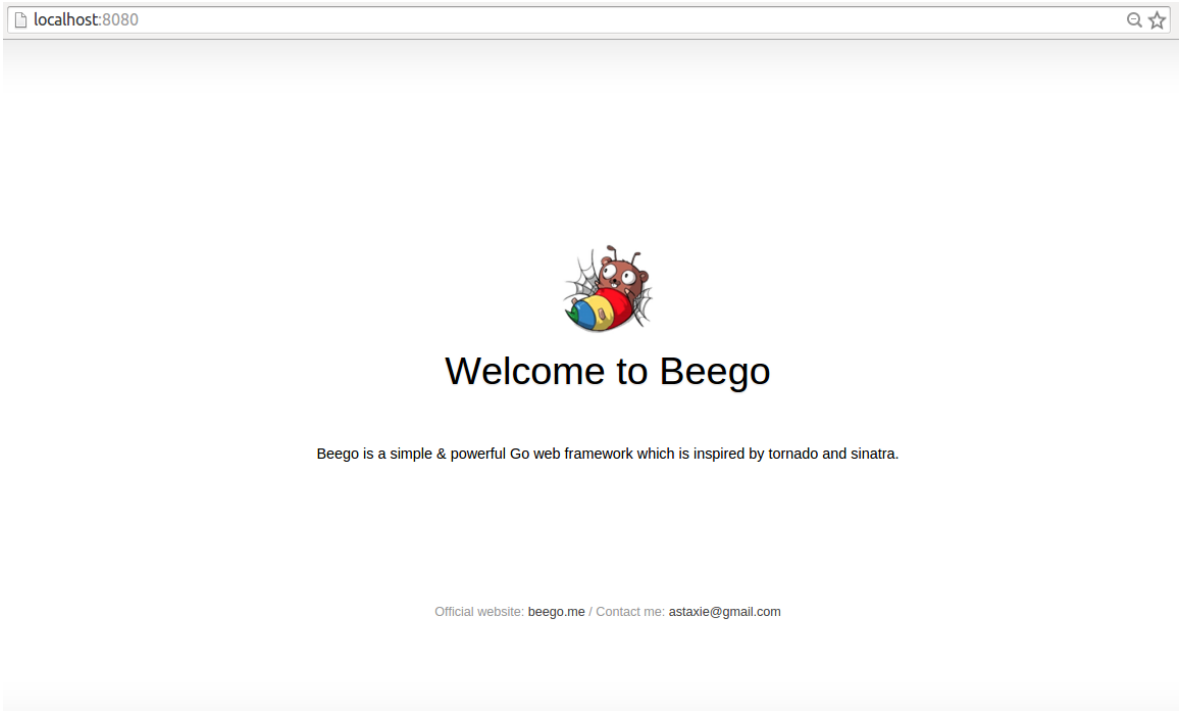
从目录结构中我们也可以看出来这是一个典型的 MVC 架构的应用，`main.go` 是入口文件。

运行项目

beego 项目创建之后，我们就开始运行项目，首先进入创建的项目，我们使用 `bee run` 来运行该项目，这样就可以做到热编译的效果：

```
➜  src  cd quickstart
➜  quickstart  bee run
2014/11/06 18:18:34 [INFO] Uses 'quickstart' as 'appname'
2014/11/06 18:18:34 [INFO] Initializing watcher...
2014/11/06 18:18:34 [TRAC] Directory(/gopath/src/quickstart/controllers)
2014/11/06 18:18:34 [TRAC] Directory(/gopath/src/quickstart)
2014/11/06 18:18:34 [TRAC] Directory(/gopath/src/quickstart/routers)
2014/11/06 18:18:34 [TRAC] Directory(/gopath/src/quickstart/tests)
2014/11/06 18:18:34 [INFO] Start building...
2014/11/06 18:18:35 [SUCC] Build was successful
2014/11/06 18:18:35 [INFO] Restarting quickstart ...
2014/11/06 18:18:35 [INFO] ./quickstart is running...
2014/11/06 18:18:35 [app.go:96] [I] http server Running on :8080
```

这样我们的应用已经在 `8080` 端口(beego 的默认端口)跑起来了.你是不是觉得很神奇，为什么没有 nginx 和 apache 居然可以自己干这个事情？是的，Go 其实已经做了网络层的东西，beego 只是封装了一下，所以可以做到不需要 nginx 和 apache。让我们打开浏览器看看效果吧：


localhost:8080
Welcome to Beego
Beego is a simple & powerful Go web framework which is inspired by tornado and sinatra.
Official website: beego.me / Contact me: astaxie@gmail.com

路由设置

路由设置

项目路由设置

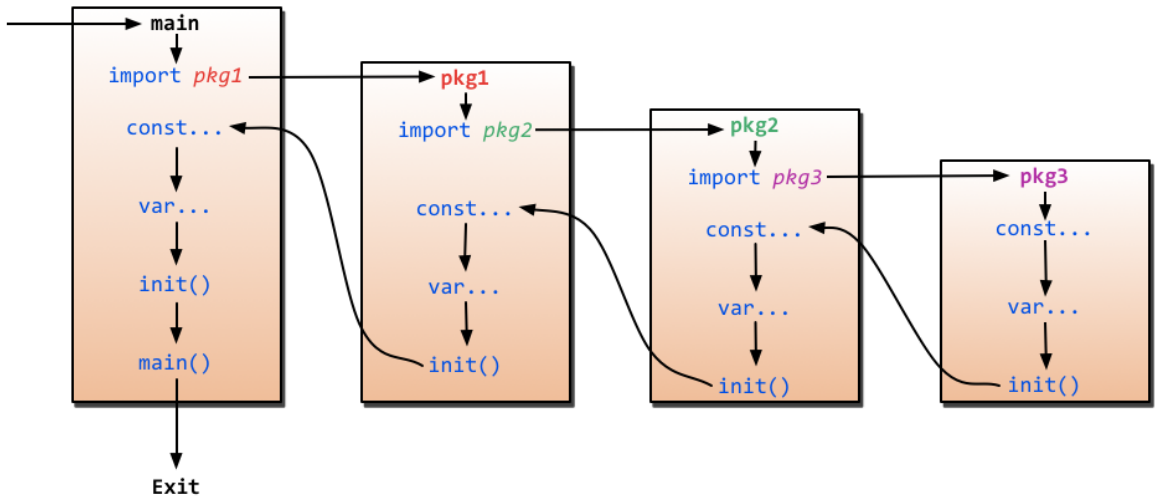
前面我们已经创建了 beego 项目，而且我们也看到它已经运行起来了，那么是如何运行起来的呢？让我们从入口文件先分析起来吧：

```
package main

import (
    _ "quickstart/routers"
    "github.com/astaxie/beego"
)

func main() {
    beego.Run()
}
```

我们看到 main 函数是入口函数，但是我们知道 Go 的执行过程是如下图所示的方式：



这里我们就看到了我们引入了一个包 `_ "quickstart/routers"` ,这个包只引入执行了里面的 init 函数，那么让我们看看这个里面做了什么事情：

```
package routers

import (
    "quickstart/controllers"
    "github.com/astaxie/beego"
)

func init() {
    beego.Router("/", &controllers.MainController{})
}
```

路由包里面我们看到执行了路由注册 `beego.Router`，这个函数的功能是映射 URL 到 controller，第一个参数是 URL (用户请求的地址)，这里我们注册的是 `/`，也就是我们访问的不带任何参数的 URL，第二个参数是对应的 Controller，也就是我们即将把请求分发到那个控制器来执行相应的逻辑，我们可以执行类似的方式注册如下路由：

```
beego.Router("/user", &controllers.UserController{})
```

这样用户就可以通过访问 `/user` 去执行 `UserController` 的逻辑。这就是我们所谓的路由，更多更复杂的路由规则请查询beego 的路由设置

再回来看看 main 函数里面的 `beego.Run`， `beego.Run` 执行之后，我们看到的效果好像只是监听服务端口这个过程，但是它内部做了很多事情：

- 解析配置文件

  beego 会自动解析在 conf 目录下面的配置文件 `app.conf`，通过修改配置文件相关的属性，我们可以定义：开启的端口，是否开启 session，应用名称等信息。

- 执行用户的 hookfunc

  beego 会执行用户注册的 hookfunc，默认的已经存在了注册 mime，用户可以通过函数 `AddAPPStartHook` 注册自己的启动函数。

- 是否开启 session

  会根据上面配置文件的分析之后判断是否开启 session，如果开启的话就初始化全局的 session。

- 是否编译模板

beego 会在启动的时候根据配置把 views 目录下的所有模板进行预编译，然后存在 map 里面，这样可以有效的提高模板运行的效率，无需进行多次编译。

- 是否开启文档功能

  根据 EnableDocs 配置判断是否开启内置的文档路由功能

- 是否启动管理模块

  beego 目前做了一个很酷的模块，应用内监控模块，会在 8088 端口做一个内部监听，我们可以通过这个端口查询到 QPS、CPU、内存、GC、goroutine、thread 等统计信息。

- 监听服务端口

  这是最后一步也就是我们看到的访问 8080 看到的网页端口，内部其实调用了 `ListenAndServe` ，充分利用了 goroutine 的优势

一旦 run 起来之后，我们的服务就监听在两个端口了，一个服务端口 8080 作为对外服务，另一个 8088 端口实行对内监控。

通过这个代码的分析我们了解了 beego 运行起来的过程，以及内部的一些机制。接下来让我们去剥离 Controller 如何来处理逻辑的。

Controller运行机制

Controller运行机制

controller 逻辑

前面我们了解了如何把用户的请求分发到控制器，这小节我们就介绍大家如何来写控制器，首先我们还是从源码分析入手：

```
package controllers

import (
        "github.com/astaxie/beego"
)

type MainController struct {
        beego.Controller
}

func (this *MainController) Get() {
        this.Data["Website"] = "beego.me"
        this.Data["Email"] = "astaxie@gmail.com"
        this.TplName = "index.tpl"
}
```

上面的代码显示首先我们声明了一个控制器 `MainController` ，这个控制器里面内嵌了 `beego.Controller` ，这就是 Go 的嵌入方式，也就是 `MainController` 自动拥有了所有 `beego.Controller` 的方法。

而 `beego.Controller` 拥有很多方法，其中包括 `Init` 、 `Prepare` 、 `Post` 、 `Get` 、 `Delete` 、 `Head` 等方法。我们可以通过重写的方式来实现这些方法，而我们上面的代码就是重写了 `Get` 方法。

我们先前介绍过 beego 是一个 RESTful 的框架，所以我们的请求默认是执行对应 `req.Method` 的方法。例如浏览器的是 `GET` 请求，那么默认就会执行 `MainController` 下的 `Get` 方法。这样我们上面的 Get 方法就会被执行到，这样就进入了我们的逻辑处理。（用户可以改变这个行为，通过注册自定义的函数名

里面的代码是需要执行的逻辑，这里只是简单的输出数据，我们可以通过各种方式获取数据，然后赋值到 `this.Data` 中，这是一个用来存储输出数据的 map，可以赋值任意类型的

值，这里我们只是简单举例输出两个字符串。

最后一个就是需要去渲染的模板，`this.TplName` 就是需要渲染的模板，这里指定了 `index.tpl`，如果用户不设置该参数，那么默认会去到模板目录的 `Controller/<方法名>.tpl` 查找，例如上面的方法会去 `maincontroller/get.tpl` ***(文件、文件夹必须小写)***。

用户设置了模板之后系统会自动的调用 `Render` 函数（这个函数是在 beego.Controller 中实现的），所以无需用户自己来调用渲染。

当然也可以不使用模版，直接用 `this.Ctx.WriteString` 输出字符串，如：

```
func (this *MainController) Get() {
        this.Ctx.WriteString("hello")
}
```

Model逻辑

Model逻辑

model 分析

我们知道 Web 应用中我们用的最多的就是数据库操作，而 model 层一般用来做这些操作，我们的 `bee new` 例子不存在 Model 的演示，但是 `bee api` 应用中存在 model 的应用。说的简单一点，如果您的应用足够简单，那么 Controller 可以处理一切的逻辑，如果您的逻辑里面存在着可以复用的东西，那么就抽取出来变成一个模块。因此 Model 就是逐步抽象的过程，一般我们会在 Model 里面处理一些数据读取，如下是一个日志分析应用中的代码片段：

```
package models

import (
    "loggo/utils"
    "path/filepath"
    "strconv"
    "strings"
)

var (
    NotPV []string = []string{"css", "js", "class", "gif", "jpg", "jpeg", "png",
"bmp", "ico", "rss", "xml", "swf"}
)

const big = 0xFFFFFF

func LogPV(urls string) bool {
    ext := filepath.Ext(urls)
    if ext == "" {
        return true
    }
    for _, v := range NotPV {
        if v == strings.ToLower(ext) {
            return false
        }
    }
    return true
}
```

所以如果您的应用足够简单，那么就不需要 Model 了；如果你的模块开始多了，需要复用，需要逻辑分离了，那么 Model 是必不可少的。接下来我们将分析如何编写 View 层的东西。

View编写

View编写

在前面编写 Controller 的时候，我们在 Get 里面写过这样的语句 `this.TplName = "index.tpl"` ，设置显示的模板文件，默认支持 `tpl` 和 `html` 的后缀名，如果想设置其他后缀你可以调用 `beego.AddTemplateExt` 接口设置，那么模板如何来显示相应的数据呢？beego 采用了 Go 语言默认的模板引擎，所以和 Go 的模板语法一样，Go 模板的详细使用方法请参考《Go Web 编程》模板使用指南

我们看看快速入门里面的代码（去掉了 css 样式）：

```
<!DOCTYPE html>

<html>
    <head>
        <title>Beego</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    </head>
    <body>
        <header class="hero-unit" style="background-color:#A9F16C">
            <div class="container">
                <div class="row">
                    <div class="hero-text">
                        <h1>Welcome to Beego!</h1>
                        <p class="description">
                            Beego is a simple & powerful Go web framework which
 is inspired by tornado and sinatra.
                            <br />
                            Official website: <a href="http://{{.Website}}">{{.W
ebsite}}</a>
                            <br />
                            Contact me: {{.Email}}
                        </p>
                    </div>
                </div>
            </div>
        </header>
    </body>
</html>
```

我们在 Controller 里面把数据赋值给了 data（map 类型），然后我们在模板中就直接通过 key 访问 `.Website` 和 `.Email` 。这样就做到了数据的输出。接下来我们讲解如何让静态文件输出。

静态文件处理

静态文件处理

前面我们介绍了如何输出静态页面，但是我们的网页往往包含了很多的静态文件，包括图片、JS、CSS 等，刚才创建的应用里面就创建了如下目录：

```
├── static
    |   ├── css
    |   ├── img
    |   └── js
```

beego 默认注册了 static 目录为静态处理的目录，注册样式：URL 前缀和映射的目录（在/main.go文件中beego.Run()之前加入）：

```
StaticDir["/static"] = "static"
```

用户可以设置多个静态文件处理目录，例如你有多个文件下载目录 download1、download2，你可以这样映射（在 /main.go 文件中 beego.Run() 之前加入）：

```
beego.SetStaticPath("/down1", "download1")
beego.SetStaticPath("/down2", "download2")
```

这样用户访问 URL `http://localhost:8080/down1/123.txt` 则会请求 download1 目录下的 123.txt 文件。

beego的MVC架构介绍

controller设计

model设计

view设计

controller设计

参数配置

参数配置

beego 目前支持 INI、XML、JSON、YAML 格式的配置文件解析，但是默认采用了 INI 格式解析，用户可以通过简单的配置就可以获得很大的灵活性。

默认配置解析

beego 默认会解析当前应用下的 `conf/app.conf` 文件。

通过这个文件你可以初始化很多 beego 的默认参数：

```
appname = beepkg
httpaddr = "127.0.0.1"
httpport = 9090
runmode ="dev"
autorender = false
recoverpanic = false
viewspath = "myview"
```

上面这些参数会替换 beego 默认的一些参数, beego 的参数主要有哪些呢？请参考 https://godoc.org/github.com/astaxie/beego#pkg-constants 。
BConfig 就是 beego 里面的默认的配置，你也可以直接通过 `beego.BConfig.AppName="beepkg"` 这样来修改，和上面的配置效果一样，只是一个在代码里面写死了，
而配置文件就会显得更加灵活。

你也可以在配置文件中配置应用需要用的一些配置信息，例如下面所示的数据库信息：

```
mysqluser = "root"
mysqlpass = "rootpass"
mysqlurls = "127.0.0.1"
mysqldb   = "beego"
```

那么你就可以通过如下的方式获取设置的配置信息:

```
beego.AppConfig.String("mysqluser")
beego.AppConfig.String("mysqlpass")
```

```
beego.AppConfig.String("mysqlurls")
beego.AppConfig.String("mysqldb")
```

AppConfig 的方法如下：

- Set(key, val string) error
- String(key string) string
- Strings(key string) []string
- Int(key string) (int, error)
- Int64(key string) (int64, error)
- Bool(key string) (bool, error)
- Float(key string) (float64, error)
- DefaultString(key string, defaultVal string) string
- DefaultStrings(key string, defaultVal []string)
- DefaultInt(key string, defaultVal int) int
- DefaultInt64(key string, defaultVal int64) int64
- DefaultBool(key string, defaultVal bool) bool
- DefaultFloat(key string, defaultVal float64) float64
- DIY(key string) (interface{}, error)
- GetSection(section string) (map[string]string, error)
- SaveConfigFile(filename string) error

在使用 ini 类型的配置文件中, key 支持 section::key 模式.

你可以用 Default* 方法返回默认值.

不同级别的配置

在配置文件里面支持 section，可以有不同的 Runmode 的配置，默认优先读取 runmode 下的配置信息，例如下面的配置文件：

```
appname = beepkg
httpaddr = "127.0.0.1"
httpport = 9090
runmode ="dev"
autorender = false
recoverpanic = false
viewspath = "myview"
```

```
[dev]
httpport = 8080
[prod]
httpport = 8088
[test]
httpport = 8888
```

上面的配置文件就是在不同的 runmode 下解析不同的配置，例如在 dev 模式下，httpport 是 8080，在 prod 模式下是 8088，在 test 模式下是 8888。其他配置文件同理。解析的时候优先解析 runmode 下的配置，然后解析默认的配置。

读取不同模式下配置参数的方法是“模式::配置参数名”，比如：beego.AppConfig.String(“dev::mysqluser”)。

对于自定义的参数，需使用 beego.GetConfig(typ, key string, defaultVal interface{}) 来获取指定 runmode 下的配置（需 1.4.0 以上版本），typ 为参数类型，key 为参数名，defaultVal 为默认值。

多个配置文件

INI 格式配置支持 `include` 方式，引用多个配置文件，例如下面的两个配置文件效果同上：

app.conf

```
appname = beepkg
httpaddr = "127.0.0.1"
httpport = 9090

include "app2.conf"
```

app2.conf

```
runmode ="dev"
autorender = false
recoverpanic = false
viewspath = "myview"

[dev]
httpport = 8080
[prod]
httpport = 8088
[test]
httpport = 8888
```

支持环境变量配置

配置文件解析支持从环境变量中获取配置项，配置项格式： `${环境变量}` 。例如下面的配置中优先使用环境变量中配置的 runmode 和 httpport，如果有配置环境变量 ProRunMode 则优先使用该环境变量值。如果不存在或者为空，则使用 “dev” 作为 runmode。

app.conf

```
runmode  = "${ProRunMode||dev}"
httpport = "${ProPort||9090}"
```

系统默认参数

beego 中带有很多可配置的参数，我们来一一认识一下它们，这样有利于我们在接下来的 beego 开发中可以充分的发挥他们的作用(你可以通过在 `conf/app.conf` 中设置对应的值，不区分大小写)：

基础配置

- BConfig

  保存了所有 beego 里面的系统默认参数，你可以通过 `beego.BConfig` 来访问和修改底下的所有配置信息.

> s="default">
>
> 配置文件路径，默认是应用程序对应的目录下的 `conf/app.conf` ，用户可以在程序代码中加载自己的配置文件
> `beego.LoadAppConfig("ini", "conf/app2.conf")`
> 也可以加载多个文件，只要你调用多次就可以了，如果后面的文件和前面的 key 冲突，那么以最新加载的为最新值

App 配置

- AppName

  应用名称，默认是 beego。通过 `bee new` 创建的是创建的项目名。

  `beego.BConfig.AppName = "beego"`

- RunMode

应用的运行模式，可选值为 `prod` , `dev` 或者 `test` . 默认是 `dev` , 为开发模式，在开发模式下出错会提示友好的出错页面，如前面错误描述中所述。

```
beego.BConfig.RunMode = "dev"
```

- RouterCaseSensitive

  是否路由忽略大小写匹配，默认是 true，区分大小写

  ```
  beego.BConfig.RouterCaseSensitive = true
  ```

- ServerName

  beego 服务器默认在请求的时候输出 server 为 beego。

  ```
  beego.BConfig.ServerName = "beego"
  ```

- RecoverPanic

  是否异常恢复，默认值为 true，即当应用出现异常的情况，通过 recover 恢复回来，而不会导致应用异常退出。

  ```
  beego.BConfig.RecoverPanic = true
  ```

- CopyRequestBody

  是否允许在 HTTP 请求时，返回原始请求体数据字节，默认为 false （GET or HEAD or 上传文件请求除外）。

  ```
  beego.BConfig.CopyRequestBody = false
  ```

- EnableGzip

  是否开启 gzip 支持，默认为 false 不支持 gzip，一旦开启了 gzip，那么在模板输出的内容会进行 gzip 或者 zlib 压缩，根据用户的 Accept-Encoding 来判断。

  ```
  beego.BConfig.EnableGzip = false
  ```

  Gzip允许用户自定义压缩级别、压缩长度阈值和针对请求类型压缩:

i. 压缩级别，`gzipCompressLevel = 9`,取值为 1~9,如果不设置为 1(最快压缩)

ii. 压缩长度阈值，`gzipMinLength = 256`,当原始内容长度大于此阈值时才开启压缩,默认为 20B(ngnix默认长度)

iii. 请求类型，`includedMethods = get;post`,针对哪些请求类型进行压缩,默认只针对 GET 请求压缩

- MaxMemory

  文件上传默认内存缓存大小，默认值是 `1 << 26` (64M)。

  ```
  beego.BConfig.MaxMemory = 1 << 26
  ```

- EnableErrorsShow

  是否显示系统错误信息，默认为 true。

  ```
  beego.BConfig.EnableErrorsShow = true
  ```

- EnableErrorsRender

  是否将错误信息进行渲染，默认值为 true，即出错会提示友好的出错页面，对于 API 类型的应用可能需要将该选项设置为 false 以阻止在 `dev` 模式下不必要的模板渲染信息返回。

Web配置

- AutoRender

  是否模板自动渲染，默认值为 true，对于 API 类型的应用，应用需要把该选项设置为 false，不需要渲染模板。

  ```
  beego.BConfig.WebConfig.AutoRender = true
  ```

- EnableDocs

  是否开启文档内置功能，默认是 false

  ```
  beego.BConfig.WebConfig.EnableDocs = true
  ```

- FlashName

  Flash 数据设置时 Cookie 的名称，默认是 BEEGO_FLASH

  ```
  beego.BConfig.WebConfig.FlashName = "BEEGO_FLASH"
  ```

- FlashSeperator

  Flash 数据的分隔符，默认是 BEEGOFLASH

  ```
  beego.BConfig.WebConfig.FlashSeparator = "BEEGOFLASH"
  ```

- DirectoryIndex

  是否开启静态目录的列表显示，默认不显示目录，返回 403 错误。

  ```
  beego.BConfig.WebConfig.DirectoryIndex = false
  ```

- StaticDir

  静态文件目录设置，默认是static

  可配置单个或多个目录:

  i. 单个目录，`StaticDir = download`. 相当于
  `beego.SetStaticPath("/download","download")`

  ii. 多个目录，`StaticDir = download:down download2:down2`. 相当于
  `beego.SetStaticPath("/download","down")` 和
  `beego.SetStaticPath("/download2","down2")`

  `beego.BConfig.WebConfig.StaticDir`

- StaticExtensionsToGzip

  允许哪些后缀名的静态文件进行 gzip 压缩，默认支持 .css 和 .js

  ```
  beego.BConfig.WebConfig.StaticExtensionsToGzip = []string{".css", ".js"}
  ```

  等价 config 文件中

```
StaticExtensionsToGzip = .css, .js
```

- TemplateLeft

  模板左标签，默认值是 `{{` 。

  ```
  beego.BConfig.WebConfig.TemplateLeft="{{"
  ```

- TemplateRight

  模板右标签，默认值是 `}}` 。

  ```
  beego.BConfig.WebConfig.TemplateRight="}}"
  ```

- ViewsPath

  模板路径，默认值是 views。

  ```
  beego.BConfig.WebConfig.ViewsPath="views"
  ```

- EnableXSRF

  是否开启 XSRF，默认为 false，不开启。

  ```
  beego.BConfig.WebConfig.EnableXSRF = false
  ```

- XSRFKEY

  XSRF 的 key 信息，默认值是 beegoxsrf。 EnableXSRF＝true 才有效

  ```
  beego.BConfig.WebConfig.XSRFKEY = "beegoxsrf"
  ```

- XSRFExpire

  XSRF 过期时间，默认值是 0，不过期。

  ```
  beego.BConfig.WebConfig.XSRFExpire = 0
  ```

监听配置

- Graceful

是否开启热升级，默认是 false，关闭热升级。

```
beego.BConfig.Listen.Graceful=false
```

- ServerTimeOut

  设置 HTTP 的超时时间，默认是 0，不超时。

  ```
  beego.BConfig.Listen.ServerTimeOut=0
  ```

- ListenTCP4

  监听本地网络地址类型，默认是TCP6，可以通过设置为true设置为TCP4。

  ```
  beego.BConfig.Listen.ListenTCP4 = true
  ```

- EnableHTTP

  是否启用 HTTP 监听，默认是 true。

  ```
  beego.BConfig.Listen.EnableHTTP = true
  ```

- HTTPAddr

  应用监听地址，默认为空，监听所有的网卡 IP。

  ```
  beego.BConfig.Listen.HTTPAddr = ""
  ```

- HTTPPort

  应用监听端口，默认为 8080。

  ```
  beego.BConfig.Listen.HTTPPort = 8080
  ```

- EnableHTTPS

  是否启用 HTTPS，默认是 false 关闭。当需要启用时，先设置 EnableHTTPS = true，并设置 `HTTPSCertFile` 和 `HTTPSKeyFile`

  ```
  beego.BConfig.Listen.EnableHTTPS = false
  ```

- HTTPSAddr

  应用监听地址，默认为空，监听所有的网卡 IP。

  ```
  beego.BConfig.Listen.HTTPSAddr = ""
  ```

- HTTPSPort

  应用监听端口，默认为 10443

  ```
  beego.BConfig.Listen.HTTPSPort = 10443
  ```

- HTTPSCertFile

  开启 HTTPS 后，ssl 证书路径，默认为空。

  ```
  beego.BConfig.Listen.HTTPSCertFile = "conf/ssl.crt"
  ```

- HTTPSKeyFile

  开启 HTTPS 之后，SSL 证书 keyfile 的路径。

  ```
  beego.BConfig.Listen.HTTPSKeyFile = "conf/ssl.key"
  ```

- EnableAdmin

  是否开启进程内监控模块，默认 false 关闭。

  ```
  beego.BConfig.Listen.EnableAdmin = false
  ```

- AdminAddr

  监控程序监听的地址，默认值是 localhost 。

  ```
  beego.BConfig.Listen.AdminAddr = "localhost"
  ```

- AdminPort

  监控程序监听的地址，默认值是 8088 。

  ```
  beego.BConfig.Listen.AdminPort = 8088
  ```

- EnableFcgi

  是否启用 fastcgi ， 默认是 false。

  ```
  beego.BConfig.Listen.EnableFcgi = false
  ```

- EnableStdIo

  通过fastcgi 标准I/O，启用 fastcgi 后才生效，默认 false。

  ```
  beego.BConfig.Listen.EnableStdIo = false
  ```

Session配置

- SessionOn

  session 是否开启，默认是 false。

  ```
  beego.BConfig.WebConfig.Session.SessionOn = false
  ```

- SessionProvider

  session 的引擎，默认是 memory，详细参见 `session 模块` 。

  ```
  beego.BConfig.WebConfig.Session.SessionProvider = ""
  ```

- SessionName

  存在客户端的 cookie 名称，默认值是 beegosessionID。

  ```
  beego.BConfig.WebConfig.Session.SessionName = "beegosessionID"
  ```

- SessionGCMaxLifetime

  session 过期时间，默认值是 3600 秒。

  ```
  beego.BConfig.WebConfig.Session.SessionGCMaxLifetime = 3600
  ```

- SessionProviderConfig

配置信息，根据不同的引擎设置不同的配置信息，详细的配置请看下面的引擎设置，详细参见 session 模块

- SessionCookieLifeTime

  session 默认存在客户端的 cookie 的时间，默认值是 3600 秒。

  ```
  beego.BConfig.WebConfig.Session.SessionCookieLifeTime = 3600
  ```

- SessionAutoSetCookie

  是否开启SetCookie, 默认值 true 开启。

  ```
  beego.BConfig.WebConfig.Session.SessionAutoSetCookie = true
  ```

- SessionDomain

  session cookie 存储域名, 默认空。

  ```
  beego.BConfig.WebConfig.Session.SessionDomain = ""
  ```

Log配置

log详细配置，请参见 `logs 模块`。

- AccessLogs

  是否输出日志到 Log，默认在 prod 模式下不会输出日志，默认为 false 不输出日志。此参数不支持配置文件配置。

  ```
  beego.BConfig.Log.AccessLogs = false
  ```

- FileLineNum

  是否在日志里面显示文件名和输出日志行号，默认 true。此参数不支持配置文件配置。

  ```
  beego.BConfig.Log.FileLineNum = true
  ```

- Outputs

日志输出配置，参考 logs 模块，console file 等配置，此参数不支持配置文件配置。

```
beego.BConfig.Log.Outputs = map[string]string{"console": ""}
```

or

```
beego.BConfig.Log.Outputs["console"] = ""
```

路由设置

路由设置

什么是路由设置呢？前面介绍的 MVC 结构执行时，介绍过 beego 存在三种方式的路由:固定路由、正则路由、自动路由，接下来详细的讲解如何使用这三种路由。

基础路由

从 beego 1.2 版本开始支持了基本的 RESTful 函数式路由,应用中的大多数路由都会定义在 `routers/router.go` 文件中。最简单的 beego 路由由 URI 和闭包函数组成。

基本 GET 路由

```
beego.Get("/",func(ctx *context.Context){
    ctx.Output.Body([]byte("hello world"))
})
```

基本 POST 路由

```
beego.Post("/alice",func(ctx *context.Context){
    ctx.Output.Body([]byte("bob"))
})
```

注册一个可以响应任何 HTTP 的路由

```
beego.Any("/foo",func(ctx *context.Context){
    ctx.Output.Body([]byte("bar"))
})
```

所有的支持的基础函数如下所示

- beego.Get(router, beego.FilterFunc)
- beego.Post(router, beego.FilterFunc)
- beego.Put(router, beego.FilterFunc)
- beego.Patch(router, beego.FilterFunc)

- beego.Head(router, beego.FilterFunc)
- beego.Options(router, beego.FilterFunc)
- beego.Delete(router, beego.FilterFunc)
- beego.Any(router, beego.FilterFunc)

支持自定义的 handler 实现

有些时候我们已经实现了一些 rpc 的应用,但是想要集成到 beego 中,或者其他的 httpserver 应用,集成到 beego 中来.现在可以很方便的集成:

```
s := rpc.NewServer()
s.RegisterCodec(json.NewCodec(), "application/json")
s.RegisterService(new(HelloService), "")
beego.Handler("/rpc", s)
```

`beego.Handler(router, http.Handler)` 这个函数是关键,第一个参数表示路由 URI, 第二个就是你自己实现的 `http.Handler` , 注册之后就会把所有 rpc 作为前缀的请求分发到 `http.Handler` 中进行处理.

这个函数其实还有第三个参数就是是否是前缀匹配,默认是 false, 如果设置了 true, 那么就会在路由匹配的时候前缀匹配,即 `/rpc/user` 这样的也会匹配去运行

路由参数

后面会讲到固定路由,正则路由,这些参数一样适用于上面的这些函数

RESTful Controller 路由

在介绍这三种 beego 的路由实现之前先介绍 RESTful，我们知道 RESTful 是一种目前 API 开发中广泛采用的形式，beego 默认就是支持这样的请求方法，也就是用户 Get 请求就执行 Get 方法，Post 请求就执行 Post 方法。因此默认的路由是这样 RESTful 的请求方式。

固定路由

固定路由也就是全匹配的路由，如下所示：

```
beego.Router("/", &controllers.MainController{})
beego.Router("/admin", &admin.UserController{})
beego.Router("/admin/index", &admin.ArticleController{})
beego.Router("/admin/addpkg", &admin.AddController{})
```

如上所示的路由就是我们最常用的路由方式，一个固定的路由，一个控制器，然后根据用户请求方法不同请求控制器中对应的方法，典型的 RESTful 方式。

正则路由

为了用户更加方便的路由设置，beego 参考了 sinatra 的路由实现，支持多种方式的路由：

- beego.Router(“/api/?:id”, &controllers.RController{})

  默认匹配 //例如对于URL”/api/123”可以匹配成功，此时变量”:id”值为”123”

- beego.Router(“/api/:id”, &controllers.RController{})

  默认匹配 //例如对于URL”/api/123”可以匹配成功，此时变量”:id”值为”123”，但URL”/api/“匹配失败

- beego.Router(“/api/:id([0-9]+)”, &controllers.RController{})

  自定义正则匹配 //例如对于URL”/api/123”可以匹配成功，此时变量”:id”值为”123”

- beego.Router(“/user/:username([\\w]+)”, &controllers.RController{})

  正则字符串匹配 //例如对于URL”/user/astaxie”可以匹配成功，此时变量”:username”值为”astaxie”

- beego.Router(“/download/*.*“, &controllers.RController{})

  *匹配方式 //例如对于URL”/download/file/api.xml”可以匹配成功，此时变量”:path”值为”file/api”， “:ext”值为”xml”

- beego.Router(“/download/ceshi/*”, &controllers.RController{})

  *全匹配方式 //例如对于URL”/download/ceshi/file/api.json”可以匹配成功，此时变量”:splat”值为”file/api.json”

- beego.Router(“/:id:int”, &controllers.RController{})

  int 类型设置方式，匹配 :id为int 类型，框架帮你实现了正则 ([0-9]+)

- beego.Router(“/:hi:string”, &controllers.RController{})

string 类型设置方式，匹配 :hi 为 string 类型。框架帮你实现了正则 ([\w]+)

- beego.Router("/cms_:id([0-9]+).html", &controllers.CmsController{})

  带有前缀的自定义正则 //匹配 :id 为正则类型。匹配 cms_123.html 这样的 url :id = 123

可以在 Controller 中通过如下方式获取上面的变量：

```
this.Ctx.Input.Param(":id")
this.Ctx.Input.Param(":username")
this.Ctx.Input.Param(":splat")
this.Ctx.Input.Param(":path")
this.Ctx.Input.Param(":ext")
```

自定义方法及 RESTful 规则

上面列举的是默认的请求方法名（请求的 method 和函数名一致，例如 `GET` 请求执行 `Get` 函数， `POST` 请求执行 `Post` 函数），如果用户期望自定义函数名，那么可以使用如下方式：

```
beego.Router("/",&IndexController{},"*:Index")
```

使用第三个参数，第三个参数就是用来设置对应 method 到函数名，定义如下

- `*` 表示任意的 method 都执行该函数
- 使用 httpmethod:funcname 格式来展示
- 多个不同的格式使用 `;` 分割
- 多个 method 对应同一个 funcname，method 之间通过 `,` 来分割

以下是一个 RESTful 的设计示例：

```
beego.Router("/api/list",&RestController{},"*:ListFood")
beego.Router("/api/create",&RestController{},"post:CreateFood")
beego.Router("/api/update",&RestController{},"put:UpdateFood")
beego.Router("/api/delete",&RestController{},"delete:DeleteFood")
```

以下是多个 HTTP Method 指向同一个函数的示例：

```
beego.Router("/api",&RestController{},"get,post:ApiFunc")
```

以下是不同的 method 对应不同的函数，通过；进行分割的示例：

```
beego.Router("/simple",&SimpleController{},"get:GetFunc;post:PostFunc")
```

可用的 HTTP Method：

包含以下所有的函数

```
* get: GET 请求
* post: POST 请求
* put: PUT 请求
* delete: DELETE 请求
* patch: PATCH 请求
* options: OPTIONS 请求
* head: HEAD 请求
```

如果同时存在 * 和对应的 HTTP Method，那么优先执行 HTTP Method 的方法，例如同时注册了如下所示的路由：

```
beego.Router("/simple",&SimpleController{},"`*:AllFunc;post:PostFunc`")
```

那么执行 `POST` 请求的时候，执行 `PostFunc` 而不执行 `AllFunc` 。

> s="default">
>
> s="default">
>
> 自定义函数的路由默认不支持 RESTful 的方法，也就是如果你设置了 `beego.Router("/api",&RestController{},"post:ApiFunc")` 这样的路由，如果请求的方法是 `POST` ，那么不会默认去执行 `Post` 函数。

自动匹配

用户首先需要把需要路由的控制器注册到自动路由中：

```
beego.AutoRouter(&controllers.ObjectController{})
```

那么 beego 就会通过反射获取该结构体中所有的实现方法，你就可以通过如下的方式访问到对应的方法中：

```
/object/login   调用 ObjectController 中的 Login 方法
/object/logout  调用 ObjectController 中的 Logout 方法
```

除了前缀两个 `/:controller/:method` 的匹配之外，剩下的 url beego 会帮你自动化解析为参数，保存在 `this.Ctx.Input.Params` 当中：

```
/object/blog/2013/09/12  调用 ObjectController 中的 Blog 方法，参数如下：map[0:2013 1:09 2:12]
```

方法名在内部是保存了用户设置的，例如 Login，url 匹配的时候都会转化为小写，所以，`/object/LOGIN` 这样的 `url` 也一样可以路由到用户定义的 `Login` 方法中。

现在已经可以通过自动识别出来下面类似的所有 url，都会把请求分发到 `controller` 的 `simple` 方法：

```
/controller/simple
/controller/simple.html
/controller/simple.json
/controller/simple.xml
```

可以通过 `this.Ctx.Input.Param(":ext")` 获取后缀名。

注解路由

从 beego 1.3 版本开始支持了注解路由，用户无需在 router 中注册路由，只需要 Include 相应地 controller，然后在 controller 的 method 方法上面写上 router 注释（// @router）就可以了，详细的使用请看下面的例子：

```
// CMS API
type CMSController struct {
    beego.Controller
}

func (c *CMSController) URLMapping() {
    c.Mapping("StaticBlock", c.StaticBlock)
    c.Mapping("AllBlock", c.AllBlock)
}


// @router /staticblock/:key [get]
func (this *CMSController) StaticBlock() {

}
```

```
// @router /all/:key [get]
func (this *CMSController) AllBlock() {

}
```

可以在 `router.go` 中通过如下方式注册路由：

```
beego.Include(&CMSController{})
```

beego 自动会进行源码分析，注意只会在 dev 模式下进行生成，生成的路由放在“/routers/commentsRouter.go” 文件中。

这样上面的路由就支持了如下的路由：

- GET /staticblock/:key
- GET /all/:key

其实效果和自己通过 Router 函数注册是一样的：

```
beego.Router("/staticblock/:key", &CMSController{}, "get:StaticBlock")
beego.Router("/all/:key", &CMSController{}, "get:AllBlock")
```

同时大家注意到新版本里面增加了 URLMapping 这个函数，这是新增加的函数，用户如果没有进行注册，那么就会通过反射来执行对应的函数，如果注册了就会通过 interface 来进行执行函数，性能上面会提升很多。

namespace

```
//初始化 namespace
ns :=
beego.NewNamespace("/v1",
    beego.NSCond(func(ctx *context.Context) bool {
        if ctx.Input.Domain() == "api.beego.me" {
            return true
        }
        return false
    }),
    beego.NSBefore(auth),
    beego.NSGet("/notallowed", func(ctx *context.Context) {
        ctx.Output.Body([]byte("notAllowed"))
    }),
    beego.NSRouter("/version", &AdminController{}, "get:ShowAPIVersion"),
    beego.NSRouter("/changepassword", &UserController{}),
```

```
    beego.NSNamespace("/shop",
        beego.NSBefore(sentry),
        beego.NSGet("/:id", func(ctx *context.Context) {
            ctx.Output.Body([]byte("notAllowed"))
        }),
    ),
    beego.NSNamespace("/cms",
        beego.NSInclude(
            &controllers.MainController{},
            &controllers.CMSController{},
            &controllers.BlockController{},
        ),
    ),
)
//注册 namespace
beego.AddNamespace(ns)
```

上面这个代码支持了如下这样的请求 URL

- GET /v1/notallowed
- GET /v1/version
- GET /v1/changepassword
- POST /v1/changepassword
- GET /v1/shop/123
- GET /v1/cms/ 对应 MainController、CMSController、BlockController 中得注解路由

而且还支持前置过滤,条件判断,无限嵌套 namespace

namespace 的接口如下:

- NewNamespace(prefix string, funcs ...interface{})

  初始化 namespace 对象,下面这些函数都是 namespace 对象的方法,但是强烈推荐使用 NS 开头的相应函数注册，因为这样更容易通过 gofmt 工具看的更清楚路由的级别关系

- NSCond(cond namespaceCond)

  支持满足条件的就执行该 namespace, 不满足就不执行

- NSBefore(filiterList ...FilterFunc)

- NSAfter(filiterList ...FilterFunc)

上面分别对应 beforeRouter 和 FinishRouter 两个过滤器，可以同时注册多个过滤器

- NSInclude(cList ...ControllerInterface)
- NSRouter(rootpath string, c ControllerInterface, mappingMethods ...string)
- NSGet(rootpath string, f FilterFunc)
- NSPost(rootpath string, f FilterFunc)
- NSDelete(rootpath string, f FilterFunc)
- NSPut(rootpath string, f FilterFunc)
- NSHead(rootpath string, f FilterFunc)
- NSOptions(rootpath string, f FilterFunc)
- NSPatch(rootpath string, f FilterFunc)
- NSAny(rootpath string, f FilterFunc)
- NSHandler(rootpath string, h http.Handler)
- NSAutoRouter(c ControllerInterface)
- NSAutoPrefix(prefix string, c ControllerInterface)

上面这些都是设置路由的函数,详细的使用和上面 beego 的对应函数是一样的

- NSNamespace(prefix string, params ...innnerNamespace)

嵌套其他 namespace

```
  ns :=
    beego.NewNamespace("/v1",
      beego.NSNamespace("/shop",
          beego.NSGet("/:id", func(ctx *context.Context) {
              ctx.Output.Body([]byte("shopinfo"))
          }),
      ),
      beego.NSNamespace("/order",
```

```
            beego.NSGet("/:id", func(ctx *context.Context) {
                ctx.Output.Body([]byte("orderinfo"))
            }),
        ),
        beego.NSNamespace("/crm",
            beego.NSGet("/:id", func(ctx *context.Context) {
                ctx.Output.Body([]byte("crminfo"))
            }),
        ),
    )
```

下面这些函数都是属于 `*Namespace` 对象的方法：不建议直接使用，当然效果和上面的 NS 开头的函数是一样的，只是上面的方式更优雅，写出来的代码更容易看得懂

- Cond(cond namespaceCond)

  支持满足条件的就执行该 namespace, 不满足就不执行,例如你可以根据域名来控制 namespace

- Filter(action string, filter FilterFunc)

  action 表示你需要执行的位置, before 和 after 分别表示执行逻辑之前和执行逻辑之后的 filter

- Router(rootpath string, c ControllerInterface, mappingMethods ...string)
- AutoRouter(c ControllerInterface)
- AutoPrefix(prefix string, c ControllerInterface)
- Get(rootpath string, f FilterFunc)
- Post(rootpath string, f FilterFunc)
- Delete(rootpath string, f FilterFunc)
- Put(rootpath string, f FilterFunc)
- Head(rootpath string, f FilterFunc)
- Options(rootpath string, f FilterFunc)

- Patch(rootpath string, f FilterFunc)
- Any(rootpath string, f FilterFunc)
- Handler(rootpath string, h http.Handler)

  上面这些都是设置路由的函数,详细的使用和上面 beego 的对应函数是一样的

- Namespace(ns ... `*Namespace`)

更多的例子代码:

```
//APIS
ns :=
    beego.NewNamespace("/api",
        //此处正式版时改为验证加密请求
        beego.NSCond(func(ctx *context.Context) bool {
            if ua := ctx.Input.Request.UserAgent(); ua != "" {
                return true
            }
            return false
        }),
        beego.NSNamespace("/ios",
            //CRUD Create(创建)、Read(读取)、Update(更新)和Delete(删除)
            beego.NSNamespace("/create",
                // /api/ios/create/node/
                beego.NSRouter("/node", &apis.CreateNodeHandler{}),
                // /api/ios/create/topic/
                beego.NSRouter("/topic", &apis.CreateTopicHandler{}),
            ),
            beego.NSNamespace("/read",
                beego.NSRouter("/node", &apis.ReadNodeHandler{}),
                beego.NSRouter("/topic", &apis.ReadTopicHandler{}),
            ),
            beego.NSNamespace("/update",
                beego.NSRouter("/node", &apis.UpdateNodeHandler{}),
                beego.NSRouter("/topic", &apis.UpdateTopicHandler{}),
            ),
            beego.NSNamespace("/delete",
                beego.NSRouter("/node", &apis.DeleteNodeHandler{}),
                beego.NSRouter("/topic", &apis.DeleteTopicHandler{}),
            )
        ),
    )
```

```
beego.AddNamespace(ns)
```

控制器函数

控制器函数

> s="default">
>
> 提示：在 v1.6 中，此文档所涉及的 API 有重大变更，`this.ServeJson()` 更改为 `this.ServeJSON()`，`this.TplNames` 更改为 `this.TplName`。

基于 beego 的 Controller 设计，只需要匿名组合 `beego.Controller` 就可以了，如下所示：

```
type xxxController struct {
    beego.Controller
}
```

`beego.Controller` 实现了接口 `beego.ControllerInterface`，`beego.ControllerInterface` 定义了如下函数：

- Init(ct *context.Context, childName string, app interface{})

  这个函数主要初始化了 Context、相应的 Controller 名称，模板名，初始化模板参数的容器 Data，app 即为当前执行的 Controller 的 reflecttype，这个 app 可以用来执行子类的方法。

- Prepare()

  这个函数主要是为了用户扩展用的，这个函数会在下面定义的这些 Method 方法之前执行，用户可以重写这个函数实现类似用户验证之类。

- Get()

  如果用户请求的 HTTP Method 是 GET，那么就执行该函数，默认是 405，用户继承的子 struct 中可以实现了该方法以处理 Get 请求。

- Post()

如果用户请求的 HTTP Method 是 POST，那么就执行该函数，默认是 405，用户继承的子 struct 中可以实现了该方法以处理 Post 请求。

- Delete()

  如果用户请求的 HTTP Method 是 DELETE，那么就执行该函数，默认是 405，用户继承的子 struct 中可以实现了该方法以处理 Delete 请求。

- Put()

  如果用户请求的 HTTP Method 是 PUT，那么就执行该函数，默认是 405，用户继承的子 struct 中可以实现了该方法以处理 Put 请求.

- Head()

  如果用户请求的 HTTP Method 是 HEAD，那么就执行该函数，默认是 405，用户继承的子 struct 中可以实现了该方法以处理 Head 请求。

- Patch()

  如果用户请求的 HTTP Method 是 PATCH，那么就执行该函数，默认是 405，用户继承的子 struct 中可以实现了该方法以处理 Patch 请求.

- Options()

  如果用户请求的HTTP Method是OPTIONS，那么就执行该函数，默认是 405，用户继承的子 struct 中可以实现了该方法以处理 Options 请求。

- Finish()

  这个函数是在执行完相应的 HTTP Method 方法之后执行的，默认是空，用户可以在子 struct 中重写这个函数，执行例如数据库关闭，清理数据之类的工作。

- Render() error

  这个函数主要用来实现渲染模板，如果 beego.AutoRender 为 true 的情况下才会执行。

所以通过子 struct 的方法重写，用户就可以实现自己的逻辑，接下来我们看一个实际的例子：

```
type AddController struct {
    beego.Controller
}

func (this *AddController) Prepare() {

}

func (this *AddController) Get() {
    this.Data["content"] = "value"
    this.Layout = "admin/layout.html"
    this.TplName = "admin/add.tpl"
}

func (this *AddController) Post() {
    pkgname := this.GetString("pkgname")
    content := this.GetString("content")
    pk := models.GetCruPkg(pkgname)
    if pk.Id == 0 {
        var pp models.PkgEntity
        pp.Pid = 0
        pp.Pathname = pkgname
        pp.Intro = pkgname
        models.InsertPkg(pp)
        pk = models.GetCruPkg(pkgname)
    }
    var at models.Article
    at.Pkgid = pk.Id
    at.Content = content
    models.InsertArticle(at)
    this.Ctx.Redirect(302, "/admin/index")
}
```

从上面的例子可以看出来，通过重写方法可以实现对应 method 的逻辑，实现 RESTful 结构的逻辑处理。

下面我们再来看一种比较流行的架构，首先实现一个自己的基类 baseController，实现一些初始化的方法，然后其他所有的逻辑继承自该基类：

```
type NestPreparer interface {
        NestPrepare()
}

// baseRouter implemented global settings for all other routers.
type baseController struct {
```

```
        beego.Controller
        i18n.Locale
        user    models.User
        isLogin bool
}
// Prepare implemented Prepare method for baseRouter.
func (this *baseController) Prepare() {

        // page start time
        this.Data["PageStartTime"] = time.Now()

        // Setting properties.
        this.Data["AppDescription"] = utils.AppDescription
        this.Data["AppKeywords"] = utils.AppKeywords
        this.Data["AppName"] = utils.AppName
        this.Data["AppVer"] = utils.AppVer
        this.Data["AppUrl"] = utils.AppUrl
        this.Data["AppLogo"] = utils.AppLogo
        this.Data["AvatarURL"] = utils.AvatarURL
        this.Data["IsProMode"] = utils.IsProMode

        if app, ok := this.AppController.(NestPreparer); ok {
                app.NestPrepare()
        }
}
```

上面定义了基类，大概是初始化了一些变量，最后有一个 Init 函数中那个 app 的应用，判断当前运行的 Controller 是否是 NestPreparer 实现，如果是的话调用子类的方法，下面我们来看一下 NestPreparer 的实现：

```
type BaseAdminRouter struct {
    baseController
}

func (this *BaseAdminRouter) NestPrepare() {
    if this.CheckActiveRedirect() {
            return
    }

    // if user isn't admin, then logout user
    if !this.user.IsAdmin {
            models.LogoutUser(&this.Controller)

            // write flash message
            this.FlashWrite("NotPermit", "true")
```

```
            this.Redirect("/login", 302)
            return
    }

    // current in admin page
    this.Data["IsAdmin"] = true

    if app, ok := this.AppController.(ModelPreparer); ok {
            app.ModelPrepare()
            return
    }
}

func (this *BaseAdminRouter) Get(){
    this.TplName = "Get.tpl"
}

func (this *BaseAdminRouter) Post(){
    this.TplName = "Post.tpl"
}
```

这样我们的执行器执行的逻辑是这样的，首先执行 Prepare，这个就是 Go 语言中 struct 中寻找方法的顺序，依次往父类寻找。执行 `BaseAdminRouter` 时，查找他是否有 `Prepare` 方法，没有就寻找 `baseController`，找到了，那么就执行逻辑，然后在 `baseController` 里面的 `this.AppController` 即为当前执行的控制器 `BaseAdminRouter`，因为会执行 `BaseAdminRouter.NestPrepare` 方法。然后开始执行相应的 Get 方法或者 Post 方法。

提前终止运行

我们应用中经常会遇到这样的情况，在 Prepare 阶段进行判断，如果用户认证不通过，就输出一段信息，然后直接中止进程，之后的 Post、Get 之类的不再执行，那么如何终止呢？可以使用 `StopRun` 来终止执行逻辑，可以在任意的地方执行。

```
type RController struct {
    beego.Controller
}

func (this *RController) Prepare() {
    this.Data["json"] = map[string]interface{}{"name": "astaxie"}
    this.ServeJSON()
    this.StopRun()
}
```

> s="default">
>
> s="default">
>
> 调用 StopRun 之后，如果你还定义了 Finish 函数就不会再执行，如果需要释放资源，那么请自己在调用 StopRun 之前手工调用 Finish 函数。

在表单中使用 PUT 方法

首先要说明, 在 XHTML 1.x 标准中, 表单只支持 GET 或者 POST 方法. 虽然说根据标准, 你不应该将表单提交到 PUT 方法, 但是如果你真想的话, 也很容易, 通常可以这么做:

首先表单本身还是使用 POST 方法提交, 但是可以在表单中添加一个隐藏字段:

```
<form method="post" ...>
  <input type="hidden" name="_method" value="put" />
```

接着在 Beego 中添加一个过滤器来判断是否将请求当做 PUT 来解析:

```
var FilterMethod = func(ctx *context.Context) {
    if ctx.BeegoInput.Query("_method")!="" && ctx.BeegoInput.IsPost(){
          ctx.Request.Method = ctx.BeegoInput.Query("_method")
    }
}

beego.InsertFilter("*", beego.BeforeRouter, FilterMethod)
```

XSRF过滤

XSRF过滤

跨站请求伪造

跨站请求伪造(Cross-site request forgery)， 简称为 XSRF，是 Web 应用中常见的一个安全问题。前面的链接也详细讲述了 XSRF 攻击的实现方式。

当前防范 XSRF 的一种通用的方法，是对每一个用户都记录一个无法预知的 cookie 数据，然后要求所有提交的请求（POST/PUT/DELETE）中都必须带有这个 cookie 数据。如果此数据不匹配，那么这个请求就可能是被伪造的。

beego 有内建的 XSRF 的防范机制，要使用此机制，你需要在应用配置文件中加上 `enablexsrf` 设定：

```
enablexsrf = true
xsrfkey = 61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o
xsrfexpire = 3600
```

或者直接在 main 入口处这样设置：

```
beego.EnableXSRF = true
beego.XSRFKEY = "61oETzKXQAGaYdkL5gEmGeJJFuYh7EQnp2XdTP1o"
beego.XSRFExpire = 3600  //过期时间，默认1小时
```

如果开启了 XSRF，那么 beego 的 Web 应用将对所有用户设置一个 `_xsrf` 的 cookie 值（默认过期 1 小时），如果 `POST PUT DELET` 请求中没有这个 cookie 值，那么这个请求会被直接拒绝。如果你开启了这个机制，那么在所有被提交的表单中，你都需要加上一个域来提供这个值。你可以通过在模板中使用 专门的函数 `XSRFFormHTML()` 来做到这一点：

过期时间上面我们设置了全局的过期时间 `beego.XSRFExpire`，但是有些时候我们也可以在控制器中修改这个过期时间，专门针对某一类处理逻辑：

```
func (this *HomeController) Get(){
    this.XSRFExpire = 7200
    this.Data["xsrfdata"]=template.HTML(this.XSRFFormHTML())
}
```

在表单中使用

在 Controller 中这样设置数据：

```
func (this *HomeController) Get(){
    this.Data["xsrfdata"]=template.HTML(this.XSRFFormHTML())
}
```

然后在模板中这样设置：

```
<form action="/new_message" method="post">
  {{ .xsrfdata }}
  <input type="text" name="message"/>
  <input type="submit" value="Post"/>
</form>
```

在 JavaScript 中使用

如果你提交的是 AJAX 的 POST 请求，你还是需要在每一个请求中通过脚本添加上 _xsrf 这个值。下面是在 AJAX 的 POST 请求，使用了 jQuery 函数来为所有请求都添加 _xsrf 值：

jQuery cookie插件：https://github.com/carhartl/jquery-cookie
base64 插件：http://phpjs.org/functions/base64_decode/

```
jQuery.postJSON = function(url, args, callback) {
   var xsrf, xsrflist;
   xsrf = $.cookie("_xsrf");
   xsrflist = xsrf.split("|");
   args._xsrf = base64_decode(xsrflist[0]);
    $.ajax({url: url, data: $.param(args), dataType: "text", type: "POST",
        success: function(response) {
        callback(eval("(" + response + ")"));
    }});
};
```

扩展 jQuery

通过扩展 ajax 给每个请求加入 xsrf 的 header

需要你在 html 里保存一个 `_xsrf` 值

```
func (this *HomeController) Get(){
    this.Data["xsrf_token"] = this.XSRFToken()
```

```
}
```

放在你的 head 中

```
<head>
    <meta name="_xsrf" content="{{.xsrf_token}}" />
</head>
```

扩展 ajax 方法，将 `_xsrf` 值加入 header，扩展后支持 jquery post/get 等内部使用了 ajax 的方法

```
var ajax = $.ajax;
$.extend({
    ajax: function(url, options) {
        if (typeof url === 'object') {
            options = url;
            url = undefined;
        }
        options = options || {};
        url = options.url;
        var xsrftoken = $('meta[name=_xsrf]').attr('content');
        var headers = options.headers || {};
        var domain = document.domain.replace(/\./ig, '\\.');
        if (!/^(http:|https:).*/.test(url) || eval('/^(http:|https:)\\/\\/(.+
\\.)*' + domain + '.*/').test(url)) {
            headers = $.extend(headers, {'X-Xsrftoken':xsrftoken});
        }
        options.headers = headers;
        return ajax(url, options);
    }
});
```

对于 PUT 和 DELETE 请求（以及不使用将 form 内容作为参数的 POST 请求）来说，你也可以在 HTTP 头中以 X-XSRFToken 这个参数传递 XSRF token。

如果你需要针对每一个请求处理器定制 XSRF 行为，你可以重写 Controller 的 CheckXSRFCookie 方法。例如你需要使用一个不支持 cookie 的 API， 你可以通过将 `CheckXSRFCookie()` 函数设空来禁用 XSRF 保护机制。然而如果 你需要同时支持 cookie 和非 cookie 认证方式，那么只要当前请求是通过 cookie 进行认证的，你就应该对其使用 XSRF 保护机制，这一点至关重要。

支持controller 级别的屏蔽

XSRF 之前是全局设置的一个参数,如果设置了那么所有的 API 请求都会进行验证,但是有些时候 API 逻辑是不需要进行验证的,因此现在支持在controller 级别设置屏蔽:

```
type AdminController struct{
    beego.Controller
}

func (a *AdminController) Prepare() {
    a.EnableXSRF = false
}
```

请求数据处理

请求数据处理

获取参数

我们经常需要获取用户传递的数据，包括 Get、POST 等方式的请求，beego 里面会自动解析这些数据，你可以通过如下方式获取数据：

- GetString(key string) string
- GetStrings(key string) []string
- GetInt(key string) (int64, error)
- GetBool(key string) (bool, error)
- GetFloat(key string) (float64, error)

使用例子如下：

```
func (this *MainController) Post() {
    jsoninfo := this.GetString("jsoninfo")
    if jsoninfo == "" {
        this.Ctx.WriteString("jsoninfo is empty")
        return
    }
}
```

如果你需要的数据可能是其他类型的，例如是 int 类型而不是 int64，那么你需要这样处理：

```
func (this *MainController) Post() {
    id := this.Input().Get("id")
    intid, err := strconv.Atoi(id)
}
```

更多其他的 request 的信息，用户可以通过 `this.Ctx.Request` 获取信息，关于该对象的属性和方法参考手册 Request。

直接解析到 struct

如果要把表单里的内容赋值到一个 struct 里，除了用上面的方法一个一个获取再赋值外，beego 提供了通过另外一个更便捷的方式，就是通过 struct 的字段名或 tag 与表单字段对应直接解析到 struct。

定义 struct：

```
type user struct {
    Id    int          `form:"-"`
    Name  interface{} `form:"username"`
    Age   int          `form:"age"`
    Email string
}
```

表单：

```
<form id="user">
    名字：<input name="username" type="text" />
    年龄：<input name="age" type="text" />
    邮箱：<input name="Email" type="text" />
    <input type="submit" value="提交" />
</form>
```

Controller 里解析：

```
func (this *MainController) Post() {
    u := user{}
    if err := this.ParseForm(&u); err != nil {
        //handle error
    }
}
```

注意：

- StructTag form 的定义和renderform方法共用一个标签
- 定义 struct 时，字段名后如果有 form 这个 tag，则会以把 form 表单里的 name 和 tag 的名称一样的字段赋值给这个字段，否则就会把 form 表单里与字段名一样的表单内容赋值给这个字段。如上面例子中，会把表单中的 username 和 age 分别赋值给 user 里的 Name 和 Age 字段，而 Email 里的内容则会赋给 Email 这个字段。
- 调用 Controller ParseForm 这个方法的时候，传入的参数必须为一个 struct 的指针，否则对 struct 的赋值不会成功并返回 `xx must be a struct pointer` 的错误。

- 如果要忽略一个字段，有两种办法，一是：字段名小写开头，二是：`form` 标签的值设置为 `-`

获取 Request Body 里的内容

在 API 的开发中，我们经常会用到 `JSON` 或 `XML` 来作为数据交互的格式，如何在 beego 中获取 Request Body 里的 JSON 或 XML 的数据呢？

1. 在配置文件里设置 `copyrequestbody = true`
2. 在 Controller 中

```
func (this *ObjectController) Post() {
    var ob models.Object
    var err error
    if err = json.Unmarshal(this.Ctx.Input.RequestBody, &ob); err == nil {
        objectid := models.AddOne(ob)
        this.Data["json"] = "{\"ObjectId\":\"" + objectid + "\"}"
    } else {
        this.Data["json"] = err.Error()
    }
    this.ServeJSON()
}
```

文件上传

在 beego 中你可以很容易的处理文件上传，就是别忘记在你的 form 表单中增加这个属性 `enctype="multipart/form-data"` ，否则你的浏览器不会传输你的上传文件。

文件上传之后一般是放在系统的内存里面，如果文件的 size 大于设置的缓存内存大小，那么就放在临时文件中，默认的缓存内存是 64M，你可以通过如下来调整这个缓存内存大小:

```
beego.MaxMemory = 1<<22
```

或者在配置文件中通过如下设置：

```
maxmemory = 1<<22
```

Beego 提供了两个很方便的方法来处理文件上传：

- GetFile(key string) (multipart.File, `*multipart.FileHeader` , error)

该方法主要用于用户读取表单中的文件名 `the_file` ，然后返回相应的信息，用户根据这些变量来处理文件上传：过滤、保存文件等。

- SaveToFile(fromfile, tofile string) error

  该方法是在 GetFile 的基础上实现了快速保存的功能

  fromfile 是提交时候的 html 表单中的 name

```
<form enctype="multipart/form-data" method="post">
    <input type="file" name="uploadname" />
    <input type="submit">
</form>
```

保存的代码例子如下：

```
func (c *FormController) Post() {
    f, h, err := c.GetFile("uploadname")
    if err != nil {
        log.Fatal("getfile err ", err)
    }
    defer f.Close()
    c.SaveToFile("uploadname", "static/upload/" + h.Filename) // 保存位置在 stat
ic/upload, 没有文件夹要先创建

}
```

数据绑定

支持从用户请求中直接数据 bind 到指定的对象,例如请求地址如下

```
?id=123&isok=true&ft=1.2&ol[0]=1&ol[1]=2&ul[]=str&ul[]=array&user.Name=astaxie
```

```
var id int
this.Ctx.Input.Bind(&id, "id")  //id ==123

var isok bool
this.Ctx.Input.Bind(&isok, "isok")  //isok ==true

var ft float64
this.Ctx.Input.Bind(&ft, "ft")  //ft ==1.2
```

```
ol := make([]int, 0, 2)
this.Ctx.Input.Bind(&ol, "ol")  //ol ==[1 2]

ul := make([]string, 0, 2)
this.Ctx.Input.Bind(&ul, "ul")  //ul ==[str array]

user struct{Name}
this.Ctx.Input.Bind(&user, "user")  //user ==Name:"astaxie"}
```

Session控制

Session控制

beego 内置了 session 模块，目前 session 模块支持的后端引擎包括 memory、cookie、file、mysql、redis、couchbase、memcache、postgres，用户也可以根据相应的 interface 实现自己的引擎。

beego 中使用 session 相当方便，只要在 main 入口函数中设置如下：

```
beego.BConfig.WebConfig.Session.SessionOn = true
```

或者通过配置文件配置如下：

```
sessionon = true
```

通过这种方式就可以开启 session，如何使用 session，请看下面的例子：

```
func (this *MainController) Get() {
    v := this.GetSession("asta")
    if v == nil {
        this.SetSession("asta", int(1))
        this.Data["num"] = 0
    } else {
        this.SetSession("asta", v.(int)+1)
        this.Data["num"] = v.(int)
    }
    this.TplName = "index.tpl"
}
```

session 有几个方便的方法：

- SetSession(name string, value interface{})
- GetSession(name string) interface{}
- DelSession(name string)
- SessionRegenerateID()
- DestroySession()

session 操作主要有设置 session、获取 session、删除 session。

当然你可以通过下面的方式自己控制这些逻辑：

```
sess:=this.StartSession()
defer sess.SessionRelease()
```

sess 对象具有如下方法：

- sess.Set()
- sess.Get()
- sess.Delete()
- sess.SessionID()
- sess.Flush()

但是我还是建议大家采用 SetSession、GetSession、DelSession 三个方法来操作，避免自己在操作的过程中资源没释放的问题。

关于 Session 模块使用中的一些参数设置：

- beego.BConfig.WebConfig.Session.SessionOn

  设置是否开启 Session，默认是 false，配置文件对应的参数名：sessionon。

- beego.BConfig.WebConfig.Session.SessionProvider

  设置 Session 的引擎，默认是 memory，目前支持还有 file、mysql、redis 等，配置文件对应的参数名：sessionprovider。

- beego.BConfig.WebConfig.Session.SessionName

  设置 cookies 的名字，Session 默认是保存在用户的浏览器 cookies 里面的，默认名是 beegosessionID，配置文件对应的参数名是：sessionname。

- beego.BConfig.WebConfig.Session.SessionGCMaxLifetime

  设置 Session 过期的时间，默认值是 3600 秒，配置文件对应的参数：sessiongcmaxlifetime。

- beego.BConfig.WebConfig.Session.SessionProviderConfig

设置对应 file、mysql、redis 引擎的保存路径或者链接地址，默认值是空，配置文件对应的参数：sessionproviderconfig。

- beego.BConfig.WebConfig.Session.SessionHashFunc

  默认值为 sha1，采用 sha1 加密算法生产 sessionid

- beego.BConfig.WebConfig.Session.SessionHashKey

  默认的 key 是 beegoserversessionkey，建议用户使用的时候修改该参数

- beego.BConfig.WebConfig.Session.SessionCookieLifeTime

  设置 cookie 的过期时间，cookie 是用来存储保存在客户端的数据。

从 beego1.1.3 版本开始移除了第三方依赖库,也就是如果你想使用 mysql、redis、couchbase、memcache、postgres 这些引擎,那么你首先需要安装

```
go get -u github.com/astaxie/beego/session/mysql
```

然后在你的 main 函数中引入该库, 和数据库的驱动引入是一样的:

```
import _ "github.com/astaxie/beego/session/mysql"
```

当 SessionProvider 为 file SessionProviderConfig 是指保存文件的目录，如下所示：

```
beego.BConfig.WebConfig.Session.SessionProvider="file"
beego.BConfig.WebConfig.Session.SessionProviderConfig = "./tmp"
```

当 SessionProvider 为 mysql 时，SessionProviderConfig 是链接地址，采用 go-sql-driver，如下所示：

```
beego.BConfig.WebConfig.Session.SessionProvider = "mysql"
beego.BConfig.WebConfig.Session.SessionProviderConfig = "username:password@protocol(address)/dbname?param=value"

需要特别注意的是，在使用 mysql 存储 session 信息的时候，需要事先在 mysql 创建表，建表语句如下
```

```
    CREATE TABLE `session` (
        `session_key` char(64) NOT NULL,
```

```
        `session_data` blob,
        `session_expiry` int(11) unsigned NOT NULL,
        PRIMARY KEY (`session_key`)
    ) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

当 SessionProvider 为 redis 时，SessionProviderConfig 是 redis 的链接地址，采用了 redigo，如下所示：

```
beego.BConfig.WebConfig.Session.SessionProvider = "redis"
beego.BConfig.WebConfig.Session.SessionProviderConfig = "127.0.0.1:6379"
```

当 SessionProvider 为 memcache 时，SessionProviderConfig 是 memcache 的链接地址，采用了 memcache，如下所示：

```
beego.BConfig.WebConfig.Session.SessionProvider = "memcache"
beego.BConfig.WebConfig.Session.SessionProviderConfig = "127.0.0.1:7080"
```

当 SessionProvider 为 postgres 时，SessionProviderConfig 是 postgres 的链接地址，采用了 postgres，如下所示：

```
beego.BConfig.WebConfig.Session.SessionProvider = "postgresql"
beego.BConfig.WebConfig.Session.SessionProviderConfig = "postgres://pqgotest:password@localhost/pqgotest?sslmode=verify-full"
```

当 SessionProvider 为 couchbase 时，SessionProviderConfig 是 couchbase 的链接地址，采用了 couchbase，如下所示：

```
beego.BConfig.WebConfig.Session.SessionProvider = "couchbase"
beego.BConfig.WebConfig.Session.SessionProviderConfig = "http://bucketname:bucketpass@myserver:8091"
```

特别注意点

因为 session 内部采用了 gob 来注册存储的对象，例如 struct，所以如果你采用了非 memory 的引擎，请自己在 main.go 的 init 里面注册需要保存的这些结构体，不然会引起应用重启之后出现无法解析的错误

过滤器

过滤器

beego 支持自定义过滤中间件，例如安全验证，强制跳转等。

过滤器函数如下所示：

```
beego.InsertFilter(pattern string, position int, filter FilterFunc, params ...bool)
```

InsertFilter 函数的三个必填参数，一个可选参数

- pattern 路由规则，可以根据一定的规则进行路由，如果你全匹配可以用 `*`
- position 执行 Filter 的地方，五个固定参数如下，分别表示不同的执行过程
  - BeforeStatic 静态地址之前
    - BeforeRouter 寻找路由之前
    - BeforeExec 找到路由之后，开始执行相应的 Controller 之前
    - AfterExec 执行完 Controller 逻辑之后执行的过滤器
    - FinishRouter 执行完逻辑之后执行的过滤器
- filter filter 函数 type FilterFunc func(*context.Context)
- params
  i. 设置 returnOnOutput 的值(默认 true), 如果在进行到此过滤之前已经有输出，是否不再继续执行此过滤器,默认设置为如果前面已有输出(参数为true)，则不再执行此过滤器
  ii. 是否重置 filters 的参数，默认是 false，因为在 filters 的 pattern 和本身的路由的 pattern 冲突的时候，可以把 filters 的参数重置，这样可以保证在后续的逻辑中获取到正确的参数，例如设置了 `/api/*` 的 filter，同时又设置了 `/api/docs/*` 的 router，那么在访问 `/api/docs/swagger/abc.js` 的时候，在执行 filters 的时候设置 `:splat` 参数为 `docs/swagger/abc.js`，但是如果不清楚 filter 的这个路由参数，就会在执行路由逻辑的时候保持 `docs/swagger/abc.js`，如果设置了 true，就会重置 `:splat` 参数.

> s="default">

> s="default">
>
> AddFilter 从beego1.3 版本开始已经废除

如下例子所示，验证用户是否已经登录，应用于全部的请求：

```
var FilterUser = func(ctx *context.Context) {
    _, ok := ctx.Input.Session("uid").(int)
    if !ok && ctx.Request.RequestURI != "/login" {
        ctx.Redirect(302, "/login")
    }
}

beego.InsertFilter("/*",beego.BeforeRouter,FilterUser)
```

> s="default">
>
> s="default">
>
> 这里需要特别注意使用 session 的 Filter 必须在 BeforeStatic 之后才能获取，因为 session 没有在这之前初始化。

还可以通过正则路由进行过滤，如果匹配参数就执行：

```
var FilterUser = func(ctx *context.Context) {
    _, ok := ctx.Input.Session("uid").(int)
    if !ok {
        ctx.Redirect(302, "/login")
    }
}
beego.InsertFilter("/user/:id([0-9]+)",beego.BeforeRouter,FilterUser)
```

过滤器实现路由

beego1.1.2 开始 Context.Input 中增加了 RunController 和 RunMethod, 这样我们就可以在执行路由查找之前,在 filter 中实现自己的路由规则.

如下示例实现了如何实现自己的路由规则:

```
var UrlManager = func(ctx *context.Context) {
    // 数据库读取全部的 url mapping 数据
    urlMapping := model.GetUrlMapping()
    for baseurl,rule:=range urlMapping {
        if baseurl == ctx.Request.RequestURI {
```

```
            ctx.Input.RunController = rule.controller
            ctx.Input.RunMethod = rule.method
            break
        }
    }
}

beego.InsertFilter("/*",beego.BeforeRouter,UrlManager)
```

Flash数据

Flash数据

这个 flash 与 Adobe/Macromedia Flash 没有任何关系。它主要用于在两个逻辑间传递临时数据，flash 中存放的所有数据会在紧接着的下一个逻辑中调用后清除。一般用于传递提示和错误消息。它适合 Post/Redirect/Get 模式。下面看使用的例子：

```
// 显示设置信息
func (c *MainController) Get() {
    flash:=beego.ReadFromRequest(&c.Controller)
    if n,ok:=flash.Data["notice"];ok{
        // 显示设置成功
        c.TplName = "set_success.html"
    }else if n,ok=flash.Data["error"];ok{
        // 显示错误
        c.TplName = "set_error.html"
    }else{
        // 不然默认显示设置页面
        c.Data["list"]=GetInfo()
        c.TplName = "setting_list.html"
    }
}

// 处理设置信息
func (c *MainController) Post() {
    flash:=beego.NewFlash()
    setting:=Settings{}
    valid := Validation{}
    c.ParseForm(&setting)
    if b, err := valid.Valid(setting);err!=nil {
        flash.Error("Settings invalid!")
        flash.Store(&c.Controller)
        c.Redirect("/setting",302)
        return
    }else if b!=nil{
        flash.Error("validation err!")
        flash.Store(&c.Controller)
        c.Redirect("/setting",302)
        return
    }
    saveSetting(setting)
```

```
    flash.Notice("Settings saved!")
    flash.Store(&c.Controller)
    c.Redirect("/setting",302)
}
```

上面的代码执行的大概逻辑是这样的：

1. Get 方法执行，因为没有 flash 数据，所以显示设置页面。
2. 用户设置信息之后点击递交，执行 Post，然后初始化一个 flash，通过验证，验证出错或者验证不通过设置 flash 的错误，如果通过了就保存设置，然后设置 flash 成功设置的信息。
3. 设置完成后跳转到 Get 请求。
4. Get 请求获取到了 Flash 信息，然后执行相应的逻辑，如果出错显示出错的页面，如果成功显示成功的页面。

默认情况下 `ReadFromRequest` 函数已经实现了读取的数据赋值给 flash，所以在你的模板里面你可以这样读取数据：

```
{{.flash.error}}
{{.flash.warning}}
{{.flash.notice}}
```

flash 对象有三个级别的设置：

- Notice 提示信息
- Warning 警告信息
- Error 错误信息

URL构建

URL构建

如果可以匹配 URL ， 那么 beego 也可以生成 URL 吗？当然可以。 UrlFor() 函数就是用于构建指定函数的 URL 的。它把对应控制器和函数名结合的字符串作为第一个参数，其余参数对应 URL 中的变量。未知变量将添加到 URL 中作为查询参数。 例如：

下面定义了一个相应的控制器

```
type TestController struct {
    beego.Controller
}

func (this *TestController) Get() {
    this.Data["Username"] = "astaxie"
    this.Ctx.Output.Body([]byte("ok"))
}

func (this *TestController) List() {
    this.Ctx.Output.Body([]byte("i am list"))
}

func (this *TestController) Params() {
    this.Ctx.Output.Body([]byte(this.Ctx.Input.Params()["0"] + this.Ctx.Input.Pa
rams()["1"] + this.Ctx.Input.Params()["2"]))
}

func (this *TestController) Myext() {
    this.Ctx.Output.Body([]byte(this.Ctx.Input.Param(":ext")))
}

func (this *TestController) GetUrl() {
    this.Ctx.Output.Body([]byte(this.UrlFor(".Myext")))
}
```

下面是我们注册的路由：

```
beego.Router("/api/list", &TestController{}, "*:List")
beego.Router("/person/:last/:first", &TestController{})
beego.AutoRouter(&TestController{})
```

那么通过方式可以获取相应的URL地址：

```
beego.URLFor("TestController.List")
// 输出 /api/list

beego.URLFor("TestController.Get", ":last", "xie", ":first", "asta")
// 输出 /person/xie/asta

beego.URLFor("TestController.Myext")
// 输出 /Test/Myext

beego.URLFor("TestController.GetUrl")
// 输出 /Test/GetUrl
```

模板中如何使用

默认情况下，beego 已经注册了 urlfor 函数，用户可以通过如下的代码进行调用

```
{{urlfor "TestController.List"}}
```

为什么不在把 URL 写死在模板中，反而要动态构建？有两个很好的理由：

1. 反向解析通常比硬编码 URL 更直观。同时，更重要的是你可以只在一个地方改变 URL，而不用到处乱找。
2. URL 创建会为你处理特殊字符的转义和 Unicode 数据，不用你操心。

多种格式数据输出

多种格式数据输出

JSON、XML、JSONP

beego 当初设计的时候就考虑了 API 功能的设计，而我们在设计 API 的时候经常是输出 JSON 或者 XML 数据，那么 beego 提供了这样的方式直接输出：

注意 struct 属性应该 为 exported Identifier
首字母应该大写

- JSON 数据直接输出：

```
func (this *AddController) Get() {
    mystruct := { ... }
    this.Data["json"] = &mystruct
    this.ServeJSON()
}
```

调用 ServeJSON 之后，会设置 `content-type` 为 `application/json` ，然后同时把数据进行 JSON 序列化输出。

- XML 数据直接输出：

```
func (this *AddController) Get() {
    mystruct := { ... }
    this.Data["xml"]=&mystruct
    this.ServeXML()
}
```

调用 ServeXML 之后，会设置 `content-type` 为 `application/xml` ，同时数据会进行 XML 序列化输出。

- jsonp 调用

```
func (this *AddController) Get() {
    mystruct := { ... }
    this.Data["jsonp"] = &mystruct
    this.ServeJSONP()
}
```

调用 ServeJSONP 之后，会设置 `content-type` 为 `application/javascript`，然后同时把数据进行 JSON 序列化，然后根据请求的 callback 参数设置 jsonp 输出。

开发模式下序列化后输出的是格式化易阅读的 JSON 或 XML 字符串；在生产模式下序列化后输出的是压缩的字符串。

表单数据验证

表单数据验证

表单验证

表单验证是用于数据验证和错误收集的模块。

安装及测试

安装：

```
go get github.com/astaxie/beego/validation
```

测试：

```
go test github.com/astaxie/beego/validation
```

示例

直接使用示例：

```
import (
    "github.com/astaxie/beego/validation"
    "log"
)

type User struct {
    Name string
    Age int
}

func main() {
    u := User{"man", 40}
    valid := validation.Validation{}
    valid.Required(u.Name, "name")
    valid.MaxSize(u.Name, 15, "nameMax")
```

```
    valid.Range(u.Age, 0, 18, "age")

    if valid.HasErrors() {
        // 如果有错误信息，证明验证没通过
        // 打印错误信息
        for _, err := range valid.Errors {
            log.Println(err.Key, err.Message)
        }
    }
    // or use like this
    if v := valid.Max(u.Age, 140, "age"); !v.Ok {
        log.Println(v.Error.Key, v.Error.Message)
    }
    // 定制错误信息
    minAge := 18
    valid.Min(u.Age, minAge, "age").Message("少儿不宜！")
    // 错误信息格式化
    valid.Min(u.Age, minAge, "age").Message("%d不禁", minAge)
}
```

通过 StructTag 使用示例：

```
import (
    "log"
    "strings"

    "github.com/astaxie/beego/validation"
)

// 验证函数写在 "valid" tag 的标签里
// 各个函数之间用分号 ";" 分隔，分号后面可以有空格
// 参数用括号 "()" 括起来，多个参数之间用逗号 "," 分开，逗号后面可以有空格
// 正则函数(Match)的匹配模式用两斜杠 "/" 括起来
// 各个函数的结果的 key 值为字段名.验证函数名
type user struct {
    Id     int
    Name   string `valid:"Required;Match(/^Bee.*/)"` // Name 不能为空并且以 Bee
 开头
    Age    int    `valid:"Range(1, 140)"` // 1 <= Age <= 140，超出此范围即为不合
法
    Email  string `valid:"Email; MaxSize(100)"` // Email 字段需要符合邮箱格式，
并且最大长度不能大于 100 个字符
    Mobile string `valid:"Mobile"` // Mobile 必须为正确的手机号
    IP     string `valid:"IP"` // IP 必须为一个正确的 IPv4 地址
}
```

```
// 如果你的 struct 实现了接口 validation.ValidFormer
// 当 StructTag 中的测试都成功时，将会执行 Valid 函数进行自定义验证
func (u *user) Valid(v *validation.Validation) {
    if strings.Index(u.Name, "admin") != -1 {
        // 通过 SetError 设置 Name 的错误信息，HasErrors 将会返回 true
        v.SetError("Name", "名称里不能含有 admin")
    }
}

func main() {
    valid := validation.Validation{}
    u := user{Name: "Beego", Age: 2, Email: "dev@beego.me"}
    b, err := valid.Valid(&u)
    if err != nil {
        // handle error
    }
    if !b {
        // validation does not pass
        // blabla...
        for _, err := range valid.Errors {
            log.Println(err.Key, err.Message)
        }
    }
}
```

StructTag 可用的验证函数：

- `Required` 不为空，即各个类型要求不为其零值
- `Min(min int)` 最小值，有效类型：`int`，其他类型都将不能通过验证
- `Max(max int)` 最大值，有效类型：`int`，其他类型都将不能通过验证
- `Range(min, max int)` 数值的范围，有效类型：`int`，他类型都将不能通过验证
- `MinSize(min int)` 最小长度，有效类型：`string slice`，其他类型都将不能通过验证
- `MaxSize(max int)` 最大长度，有效类型：`string slice`，其他类型都将不能通过验证
- `Length(length int)` 指定长度，有效类型：`string slice`，其他类型都将不能通过验证
- `Alpha` alpha字符，有效类型：`string`，其他类型都将不能通过验证
- `Numeric` 数字，有效类型：`string`，其他类型都将不能通过验证

- `AlphaNumeric` alpha 字符或数字，有效类型：`string`，其他类型都将不能通过验证
- `Match(pattern string)` 正则匹配，有效类型：`string`，其他类型都将被转成字符串再匹配(fmt.Sprintf(“%v”, obj).Match)
- `AlphaDash` alpha 字符或数字或横杠 `-_`，有效类型：`string`，其他类型都将不能通过验证
- `Email` 邮箱格式，有效类型：`string`，其他类型都将不能通过验证
- `IP` IP 格式，目前只支持 IPv4 格式验证，有效类型：`string`，其他类型都将不能通过验证
- `Base64` base64 编码，有效类型：`string`，其他类型都将不能通过验证
- `Mobile` 手机号，有效类型：`string`，其他类型都将不能通过验证
- `Tel` 固定电话号，有效类型：`string`，其他类型都将不能通过验证
- `Phone` 手机号或固定电话号，有效类型：`string`，其他类型都将不能通过验证
- `ZipCode` 邮政编码，有效类型：`string`，其他类型都将不能通过验证

API 文档

请移步 Go Walker。

错误处理

错误处理

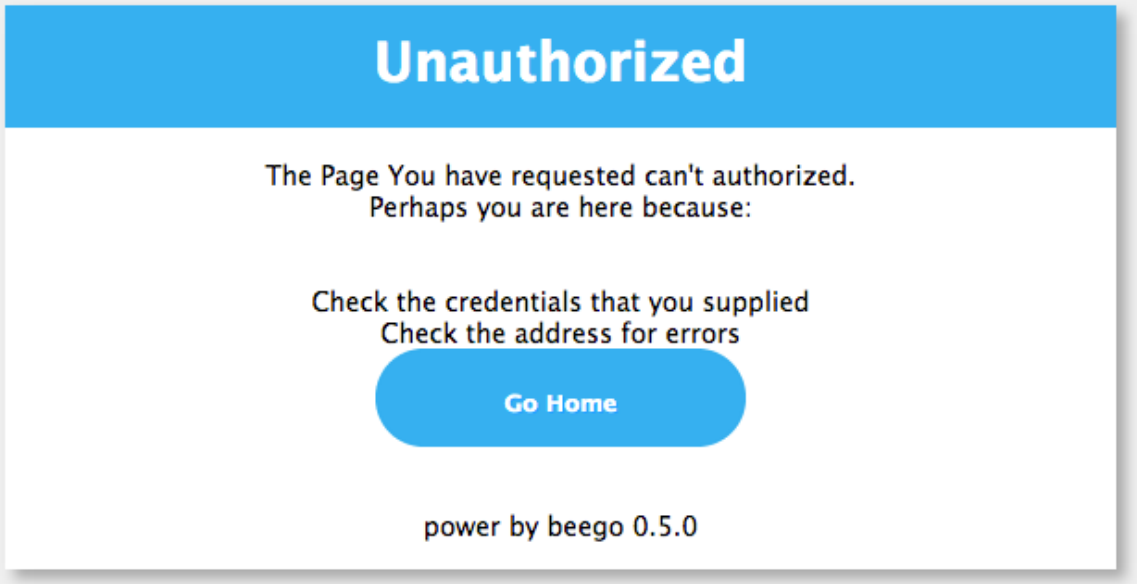
我们在做 Web 开发的时候，经常需要页面跳转和错误处理，beego 这方面也进行了考虑，通过 `Redirect` 方法来进行跳转：

```
func (this *AddController) Get() {
    this.Redirect("/", 302)
}
```

如何中止此次请求并抛出异常，beego 可以在控制器中这样操作：

```
func (this *MainController) Get() {
    this.Abort("401")
    v := this.GetSession("asta")
    if v == nil {
        this.SetSession("asta", int(1))
        this.Data["Email"] = 0
    } else {
        this.SetSession("asta", v.(int)+1)
        this.Data["Email"] = v.(int)
    }
    this.TplName = "index.tpl"
}
```

这样 `this.Abort("401")` 之后的代码不会再执行，而且会默认显示给用户如下页面：



beego 框架默认支持 401、403、404、500、503 这几种错误的处理。用户可以自定义相应的错误处理，例如下面重新定义 404 页面：

```
func page_not_found(rw http.ResponseWriter, r *http.Request){
    t,_:= template.New("404.html").ParseFiles(beego.BConfig.WebConfig.ViewsPath+"/404.html")
    data :=make(map[string]interface{})
    data["content"] = "page not found"
    t.Execute(rw, data)
}

func main() {
    beego.ErrorHandler("404",page_not_found)
    beego.Router("/", &controllers.MainController{})
    beego.Run()
}
```

我们可以通过自定义错误页面 `404.html` 来处理 404 错误。

beego 更加人性化的还有一个设计就是支持用户自定义字符串错误类型处理函数，例如下面的代码，用户注册了一个数据库出错的处理页面：

```
func dbError(rw http.ResponseWriter, r *http.Request){
    t,_:= template.New("dberror.html").ParseFiles(beego.BConfig.WebConfig.ViewsPath+"/dberror.html")
    data :=make(map[string]interface{})
    data["content"] = "database is now down"
    t.Execute(rw, data)
```

```
}

func main() {
    beego.ErrorHandler("dbError",dbError)
    beego.Router("/", &controllers.MainController{})
    beego.Run()
}
```

一旦在入口注册该错误处理代码，那么你可以在任何你的逻辑中遇到数据库错误调用 `this.Abort("dbError")` 来进行异常页面处理。

Controller 定义 Error

从 1.4.3 版本开始，支持 Controller 方式定义 Error 错误处理函数，这样就可以充分利用系统自带的模板处理，以及 context 等方法。

```
package controllers

import (
    "github.com/astaxie/beego"
)

type ErrorController struct {
    beego.Controller
}

func (c *ErrorController) Error404() {
    c.Data["content"] = "page not found"
    c.TplName = "404.tpl"
}

func (c *ErrorController) Error501() {
    c.Data["content"] = "server error"
    c.TplName = "501.tpl"
}


func (c *ErrorController) ErrorDb() {
    c.Data["content"] = "database is now down"
    c.TplName = "dberror.tpl"
}
```

通过上面的例子我们可以看到，所有的函数都是有一定规律的，都是 `Error` 开头，后面的名字就是我们调用 `Abort` 的名字，例如 `Error404` 函数其实调用对应的就是 `Abort("404")`

我们就只要在 `beego.Run` 之前采用 `beego.ErrorController` 注册这个错误处理函数就可以了

```
package main

import (
    _ "btest/routers"
    "btest/controllers"

    "github.com/astaxie/beego"
)

func main() {
    beego.ErrorController(&controllers.ErrorController{})
    beego.Run()
}
```

日志处理

日志处理

beego 之前介绍的时候说过是基于几个模块搭建的，beego 的日志处理是基于 logs 模块搭建的，内置了一个变量 `BeeLogger` ，默认已经是 `logs.BeeLogger` 类型，初始化了 console，也就是默认输出到 `console` 。

使用入门

一般在程序中我们使用如下的方式进行输出：

```
beego.Emergency("this is emergency")
beego.Alert("this is alert")
beego.Critical("this is critical")
beego.Error("this is error")
beego.Warning("this is warning")
beego.Notice("this is notice")
beego.Informational("this is informational")
beego.Debug("this is debug")
```

设置输出

我们的程序往往期望把信息输出到 log 中，现在设置输出到文件很方便，如下所示：

```
beego.SetLogger("file", `{"filename":"logs/test.log"}`)
```

更多详细的日志配置请查看日志配置

这个默认情况就会同时输出到两个地方，一个 console，一个 file，如果只想输出到文件，就需要调用删除操作：

```
beego.BeeLogger.DelLogger("console")
```

设置级别

日志的级别如上所示的代码这样分为八个级别：

```
LevelEmergency
LevelAlert
LevelCritical
LevelError
LevelWarning
LevelNotice
LevelInformational
LevelDebug
```

级别依次降低，默认全部打印，但是一般我们在部署环境，可以通过设置级别设置日志级别：

```
beego.SetLevel(beego.LevelInformational)
```

输出文件名和行号

日志默认不输出调用的文件名和文件行号,如果你期望输出调用的文件名和文件行号,可以如下设置

```
beego.SetLogFuncCall(true)
```

开启传入参数 true, 关闭传入参数 false, 默认是关闭的.

完整示例

```
func internalCalculationFunc(x, y int) (result int, err error) {
    beego.Debug("calculating z. x:", x, " y:", y)
    z := y
    switch {
    case x == 3:
        beego.Debug("x == 3")
        panic("Failure.")
    case y == 1:
        beego.Debug("y == 1")
        return 0, errors.New("Error!")
    case y == 2:
        beego.Debug("y == 2")
        z = x
    default:
        beego.Debug("default")
        z += x
    }
```

```
    retVal := z - 3
    beego.Debug("Returning ", retVal)

    return retVal, nil
}

func processInput(input inputData) {
    defer func() {
        if r := recover(); r != nil {
            beego.Error("Unexpected error occurred: ", r)
            outputs <- outputData{result: 0, error: true}
        }
    }()
    beego.Informational("Received input signal. x:", input.x, " y:", input.y)

    res, err := internalCalculationFunc(input.x, input.y)
    if err != nil {
        beego.Warning("Error in calculation:", err.Error())
    }

    beego.Informational("Returning result: ", res, " error: ", err)
    outputs <- outputData{result: res, error: err != nil}
}

func main() {
    inputs = make(chan inputData)
    outputs = make(chan outputData)
    criticalChan = make(chan int)
    beego.Informational("App started.")

    go consumeResults(outputs)
    beego.Informational("Started receiving results.")

    go generateInputs(inputs)
    beego.Informational("Started sending signals.")

    for {
        select {
        case input := <-inputs:
            processInput(input)
        case <-criticalChan:
            beego.Critical("Caught value from criticalChan: Go shut down.")
            panic("Shut down due to critical fault.")
        }
    }
}
```

model设计

概述

概述

模型（Models）— beego ORM

beego ORM 是一个强大的 Go 语言 ORM 框架。她的灵感主要来自 Django ORM 和 SQLAlchemy。

目前该框架仍处于开发阶段，可能发生任何导致不兼容的改动。

已支持数据库驱动：

- MySQL：github.com/go-sql-driver/mysql
- PostgreSQL：github.com/lib/pq
- Sqlite3：github.com/mattn/go-sqlite3

以上数据库驱动均通过基本测试，但我们仍需要您的反馈。

ORM 特性：

- 支持 Go 的所有类型存储
- 轻松上手，采用简单的 CRUD 风格
- 自动 Join 关联表
- 跨数据库兼容查询
- 允许直接使用 SQL 查询 / 映射
- 严格完整的测试保证 ORM 的稳定与健壮

更多特性请在文档中自行品读。

安装 ORM：

```
go get github.com/astaxie/beego/orm
```

修改日志

- 2016-01-18: 规范了数据库驱动的命名
- 2014-03-10: GetDB 从注册的数据库中返回 *sql.DB. ResetModelCache 重置已注册的模型struct
- 2014-02-10: 随着 beego1.1.0 的发布提交的改进
    - 关于 时区设置
    - 新增的 api:
      Ormer.InsertMulti
      Ormer.ReadOrCreate
      RawSeter.RowsToMap
      RawSeter.RowsToStruct
      orm.NewOrmWithDB
    - 改进的 api:
      RawSeter.Values 支持设置 columns
      RawSeter.ValuesList 支持设置 columns
      RawSeter.ValuesFlat 支持设置 column
      RawSeter.QueryRow/QueryRows 从对应每个strcut field位置的赋值，改为对应名称取值（不需要对应好字段数量与位置）
- 2013-10-14: 自动载入关系字段，多对多关系操作，完善关系查询
- 2013-10-09: 原子操作更新值
- 2013-09-22: RegisterDataBase maxIdle / maxConn 设置为可选参数, MySQL 自定义引擎
- 2013-09-16: 支持设置 空闲链接数 和 最大链接数 SetMaxIdleConns / SetMaxOpenConns
- 2013-09-12: Read 支持设定条件字段 Update / All / One 支持设定返回字段
- 2013-09-09: Raw SQL QueryRow/QueryRows 功能完成
- 2013-08-27: 自动建表继续改进

- 2013-08-19: 自动建表功能完成
- 2013-08-13: 更新数据库类型测试
- 2013-08-13: 增加 Go 类型支持，包括 int8、uint8、byte、rune 等
- 2013-08-13: 增强 date / datetime 的时区支持

快速入门

简单示例

```
package main

import (
    "fmt"
    "github.com/astaxie/beego/orm"
    _ "github.com/go-sql-driver/mysql" // import your used driver
)

// Model Struct
type User struct {
    Id   int
    Name string `orm:"size(100)"`
}

func init() {
    // set default database
    orm.RegisterDataBase("default", "mysql", "username:password@tcp(127.0.0.1:3306)/db_name?charset=utf8", 30)

    // register model
    orm.RegisterModel(new(User))

    // create table
    orm.RunSyncdb("default", false, true)
}

func main() {
    o := orm.NewOrm()

    user := User{Name: "slene"}
```

```
    // insert
    id, err := o.Insert(&user)
    fmt.Printf("ID: %d, ERR: %v\n", id, err)

    // update
    user.Name = "astaxie"
    num, err := o.Update(&user)
    fmt.Printf("NUM: %d, ERR: %v\n", num, err)

    // read one
    u := User{Id: user.Id}
    err = o.Read(&u)
    fmt.Printf("ERR: %v\n", err)

    // delete
    num, err = o.Delete(&u)
    fmt.Printf("NUM: %d, ERR: %v\n", num, err)
}
```

关联查询

```
type Post struct {
    Id    int    `orm:"auto"`
    Title string `orm:"size(100)"`
    User  *User  `orm:"rel(fk)"`
}

var posts []*Post
qs := o.QueryTable("post")
num, err := qs.Filter("User__Name", "slene").All(&posts)
```

SQL 查询

当您无法使用 ORM 来达到您的需求时，也可以直接使用 SQL 来完成查询 / 映射操作。

```
var maps []orm.Params
num, err := o.Raw("SELECT * FROM user").Values(&maps)
for _,term := range maps{
    fmt.Println(term["id"],":",term["name"])
}
```

事务处理

```
o.Begin()
...
user := User{Name: "slene"}
id, err := o.Insert(&user)
if err == nil {
    o.Commit()
} else {
    o.Rollback()
}
```

调试查询日志

在开发环境下，您可以使用以下指令来开启查询调试模式：

```
func main() {
    orm.Debug = true
...
```

开启后将会输出所有查询语句，包括执行、准备、事务等。

例如：

```
[ORM] - 2013-08-09 13:18:16 - [Queries/default] - [    db.Exec /     0.4ms] - [INSERT INTO `user` (`name`) VALUES (?)] - `slene`
...
```

注意：我们不建议您在部署产品后这样做。

ORM使用

ORM使用

beego/orm 的使用例子

后文例子如无特殊说明都以这个为基础。

models.go:

```
package main

import (
    "github.com/astaxie/beego/orm"
)

type User struct {
    Id          int
    Name        string
    Profile     *Profile   `orm:"rel(one)"` // OneToOne relation
    Post        []*Post `orm:"reverse(many)"` // 设置一对多的反向关系
}

type Profile struct {
    Id          int
    Age         int16
    User        *User   `orm:"reverse(one)"` // 设置一对一反向关系(可选)
}

type Post struct {
    Id    int
    Title string
    User  *User  `orm:"rel(fk)"`    //设置一对多关系
    Tags  []*Tag `orm:"rel(m2m)"`
}

type Tag struct {
    Id    int
    Name  string
    Posts []*Post `orm:"reverse(many)"` //设置多对多反向关系
}
```

```
func init() {
    // 需要在init中注册定义的model
    orm.RegisterModel(new(User), new(Post), new(Profile), new(Tag))
}
```

main.go

```
package main

import (
    "fmt"
    "github.com/astaxie/beego/orm"
    _ "github.com/go-sql-driver/mysql"
)

func init() {
    orm.RegisterDriver("mysql", orm.DRMySQL)

    orm.RegisterDataBase("default", "mysql", "root:root@/orm_test?charset=utf8")
}

func main() {
    o := orm.NewOrm()
    o.Using("default") // 默认使用 default，你可以指定为其他数据库

    profile := new(Profile)
    profile.Age = 30

    user := new(User)
    user.Profile = profile
    user.Name = "slene"

    fmt.Println(o.Insert(profile))
    fmt.Println(o.Insert(user))
}
```

数据库的设置

目前 ORM 支持三种数据库，以下为测试过的 driver

将你需要使用的 driver 加入 import 中

```
import (
    _ "github.com/go-sql-driver/mysql"
```

```
    _ "github.com/lib/pq"
    _ "github.com/mattn/go-sqlite3"
)
```

RegisterDriver

三种默认数据库类型

```
// For version 1.6
orm.DRMySQL
orm.DRSqlite
orm.DRPostgres

// < 1.6
orm.DR_MySQL
orm.DR_Sqlite
orm.DR_Postgres
```

```
// 参数1   driverName
// 参数2   数据库类型
// 这个用来设置 driverName 对应的数据库类型
// mysql / sqlite3 / postgres 这三种是默认已经注册过的，所以可以无需设置
orm.RegisterDriver("mysql", orm.DRMySQL)
```

RegisterDataBase

ORM 必须注册一个别名为 `default` 的数据库，作为默认使用。

ORM 使用 golang 自己的连接池

```
// 参数1        数据库的别名，用来在 ORM 中切换数据库使用
// 参数2        driverName
// 参数3        对应的链接字符串
orm.RegisterDataBase("default", "mysql", "root:root@/orm_test?charset=utf8")

// 参数4(可选)  设置最大空闲连接
// 参数5(可选)  设置最大数据库连接 (go >= 1.2)
maxIdle := 30
maxConn := 30
orm.RegisterDataBase("default", "mysql", "root:root@/orm_test?charset=utf8", max
Idle, maxConn)
```

SetMaxIdleConns

根据数据库的别名，设置数据库的最大空闲连接

```
orm.SetMaxIdleConns("default", 30)
```

SetMaxOpenConns

根据数据库的别名，设置数据库的最大数据库连接 (go >= 1.2)

```
orm.SetMaxOpenConns("default", 30)
```

时区设置

ORM 默认使用 time.Local 本地时区

- 作用于 ORM 自动创建的时间
- 从数据库中取回的时间转换成 ORM 本地时间

如果需要的话，你也可以进行更改

```
// 设置为 UTC 时间
orm.DefaultTimeLoc = time.UTC
```

ORM 在进行 RegisterDataBase 的同时，会获取数据库使用的时区，然后在 time.Time 类型存取时做相应转换，以匹配时间系统，从而保证时间不会出错。

注意:

- 鉴于 Sqlite3 的设计，存取默认都为 UTC 时间
- 使用 go-sql-driver 驱动时，请注意参数设置

  从某一版本开始，驱动默认使用 UTC 时间，而非本地时间，所以请指定时区参数或者全部以 UTC 时间存取

  例如：`root:root@/orm_test?charset=utf8&loc=Asia%2FShanghai`

  参见 loc / parseTime

注册模型

如果使用 orm.QuerySeter 进行高级查询的话，这个是必须的。

反之，如果只使用 Raw 查询和 map struct，是无需这一步的。您可以去查看 Raw SQL 查询

RegisterModel

将你定义的 Model 进行注册，最佳设计是有单独的 models.go 文件，在他的 init 函数中进行注册。

迷你版 models.go

```
package main

import "github.com/astaxie/beego/orm"

type User struct {
    Id   int
    Name string
}

func init(){
    orm.RegisterModel(new(User))
}
```

RegisterModel 也可以同时注册多个 model

```
orm.RegisterModel(new(User), new(Profile), new(Post))
```

详细的 struct 定义请查看文档 模型定义

RegisterModelWithPrefix

使用表名前缀

```
orm.RegisterModelWithPrefix("prefix_", new(User))
```

创建后的表名为 prefix_user

NewOrmWithDB

有时候需要自行管理连接池与数据库链接（比如：go 的链接池无法让两次查询使用同一个链接的）

但又想使用 ORM 的查询功能

```
var driverName, aliasName string
// driverName 是驱动的名称
// aliasName 是当前 db 的自定义别名
```

```
var db *sql.DB
...
o := orm.NewOrmWithDB(driverName, aliasName, db)
```

GetDB

从已注册的数据库返回 *sql.DB 对象，默认返回别名为 default 的数据库。

```
db, err := orm.GetDB()
if err != nil {
    fmt.Println("get default DataBase")
}

db, err := orm.GetDB("alias")
if err != nil {
    fmt.Println("get alias DataBase")
}
```

ResetModelCache

重置已经注册的模型 struct，一般用于编写测试用例

```
orm.ResetModelCache()
```

ORM 接口使用

使用 ORM 必然接触的 Ormer 接口，我们来熟悉一下

```
var o orm.Ormer
o = orm.NewOrm() // 创建一个 Ormer
// NewOrm 的同时会执行 orm.BootStrap (整个 app 只执行一次)，用以验证模型之间的定义并缓存。
```

切换数据库，或者，进行事务处理，都会作用于这个 Ormer 对象，以及其进行的任何查询。

所以：需要 **切换数据库** 和 **事务处理** 的话，不要使用全局保存的 Ormer 对象。

- type Ormer interface {
    - Read(interface{}, ...string) error
    - ReadOrCreate(interface{}, string, ...string) (bool, int64, error)
    - Insert(interface{}) (int64, error)

    - InsertMulti(int, interface{}) (int64, error)
    - Update(interface{}, ...string) (int64, error)
    - Delete(interface{}) (int64, error)
    - LoadRelated(interface{}, string, ...interface{}) (int64, error)
    - QueryM2M(interface{}, string) QueryM2Mer
    - QueryTable(interface{}) QuerySeter
    - Using(string) error
    - Begin() error
    - Commit() error
    - Rollback() error
    - Raw(string, ...interface{}) RawSeter
    - Driver() Driver
- }

QueryTable

传入表名，或者 Model 对象，返回一个 QuerySeter

```
o := orm.NewOrm()
var qs orm.QuerySeter
qs = o.QueryTable("user")
// 如果表没有定义过，会立刻 panic
```

Using

切换为其他数据库

```
orm.RegisterDataBase("db1", "mysql", "root:root@/orm_db2?charset=utf8")
orm.RegisterDataBase("db2", "sqlite3", "data.db")

o1 := orm.NewOrm()
o1.Using("db1")

o2 := orm.NewOrm()
o2.Using("db2")

// 切换为其他数据库以后
// 这个 Ormer 对象的其下的 api 调用都将使用这个数据库
```

默认使用 `default` 数据库，无需调用 Using

Raw

使用 sql 语句直接进行操作

Raw 函数，返回一个 RawSeter 用以对设置的 sql 语句和参数进行操作

```
o := orm.NewOrm()
var r orm.RawSeter
r = o.Raw("UPDATE user SET name = ? WHERE name = ?", "testing", "slene")
```

Driver

返回当前 ORM 使用的 db 信息

```
type Driver interface {
    Name() string
    Type() DriverType
}
```

```
orm.RegisterDataBase("db1", "mysql", "root:root@/orm_db2?charset=utf8")
orm.RegisterDataBase("db2", "sqlite3", "data.db")

o1 := orm.NewOrm()
o1.Using("db1")
dr := o1.Driver()
fmt.Println(dr.Name() == "db1") // true
fmt.Println(dr.Type() == orm.DRMySQL) // true

o2 := orm.NewOrm()
o2.Using("db2")
dr = o2.Driver()
fmt.Println(dr.Name() == "db2") // true
fmt.Println(dr.Type() == orm.DRSqlite) // true
```

调试模式打印查询语句

简单的设置 Debug 为 true 打印查询的语句

可能存在性能问题，不建议使用在生产模式

```
func main() {
    orm.Debug = true
...
```

默认使用 os.Stderr 输出日志信息

改变输出到你自己的 io.Writer

```
var w io.Writer
...
// 设置为你的 io.Writer
...
orm.DebugLog = orm.NewLog(w)
```

日志格式

```
[ORM] - 时间 - [Queries/数据库名] - [执行操作/执行时间] - [SQL语句] - 使用标点 `,` 分隔的参数列表 - 打印遇到的错误
```

```
[ORM] - 2013-08-09 13:18:16 - [Queries/default] - [    db.Exec /     0.4ms] - [INSERT INTO `user` (`name`) VALUES (?)] - `slene`
[ORM] - 2013-08-09 13:18:16 - [Queries/default] - [    db.Exec /     0.5ms] - [UPDATE `user` SET `name` = ? WHERE `id` = ?] - `astaxie`, `14`
[ORM] - 2013-08-09 13:18:16 - [Queries/default] - [db.QueryRow /     0.4ms] - [SELECT `id`, `name` FROM `user` WHERE `id` = ?] - `14`
[ORM] - 2013-08-09 13:18:16 - [Queries/default] - [    db.Exec /     0.4ms] - [INSERT INTO `post` (`user_id`,`title`,`content`) VALUES (?, ?, ?)] - `14`, `beego orm`, `powerful amazing`
[ORM] - 2013-08-09 13:18:16 - [Queries/default] - [   db.Query /     0.4ms] - [SELECT T1.`name` `User__Name`, T0.`user_id` `User`, T1.`id` `User__Id` FROM `post` T0 INNER JOIN `user` T1 ON T1.`id` = T0.`user_id` WHERE T0.`id` = ? LIMIT 1000] - `68`
[ORM] - 2013-08-09 13:18:16 - [Queries/default] - [    db.Exec /     0.4ms] - [DELETE FROM `user` WHERE `id` = ?] - `14`
[ORM] - 2013-08-09 13:18:16 - [Queries/default] - [   db.Query /     0.3ms] - [SELECT T0.`id` FROM `post` T0 WHERE T0.`user_id` IN (?) ] - `14`
[ORM] - 2013-08-09 13:18:16 - [Queries/default] - [    db.Exec /     0.4ms] - [DELETE FROM `post` WHERE `id` IN (?)] - `68`
```

日志内容包括 **所有的数据库操作**，事务，Prepare，等

CRUD操作

CRUD操作

对象的 CRUD 操作

如果已知主键的值，那么可以使用这些方法进行 CRUD 操作

对 object 操作的四个方法 Read / Insert / Update / Delete

```
o := orm.NewOrm()
user := new(User)
user.Name = "slene"

fmt.Println(o.Insert(user))

user.Name = "Your"
fmt.Println(o.Update(user))
fmt.Println(o.Read(user))
fmt.Println(o.Delete(user))
```

如果需要通过条件查询获取对象，请参见高级查询

Read

```
o := orm.NewOrm()
user := User{Id: 1}

err := o.Read(&user)

if err == orm.ErrNoRows {
    fmt.Println("查询不到")
} else if err == orm.ErrMissPK {
    fmt.Println("找不到主键")
} else {
    fmt.Println(user.Id, user.Name)
}
```

Read 默认通过查询主键赋值，可以使用指定的字段进行查询：

```
user := User{Name: "slene"}
err = o.Read(&user, "Name")
...
```

对象的其他字段值将会是对应类型的默认值

复杂的单个对象查询参见 One

ReadOrCreate

尝试从数据库读取，不存在的话就创建一个

默认必须传入一个参数作为条件字段，同时也支持多个参数多个条件字段

```
o := orm.NewOrm()
user := User{Name: "slene"}
// 三个返回参数依次为：是否新创建的，对象 Id 值，错误
if created, id, err := o.ReadOrCreate(&user, "Name"); err == nil {
    if created {
        fmt.Println("New Insert an object. Id:", id)
    } else {
        fmt.Println("Get an object. Id:", id)
    }
}
```

Insert

第一个返回值为自增健 Id 的值

```
o := orm.NewOrm()
var user User
user.Name = "slene"
user.IsActive = true

id, err := o.Insert(&user)
if err == nil {
    fmt.Println(id)
}
```

创建后会自动对 auto 的 field 赋值

InsertMulti

同时插入多个对象

类似sql语句

```
insert into table (name, age) values("slene", 28),("astaxie", 30),("unknown", 20
)
```

第一个参数 bulk 为并列插入的数量，第二个为对象的slice

返回值为成功插入的数量

```
users := []User{
    {Name: "slene"},
    {Name: "astaxie"},
    {Name: "unknown"},
    ...
}
successNums, err := o.InsertMulti(100, users)
```

bulk 为 1 时，将会顺序插入 slice 中的数据

Update

第一个返回值为影响的行数

```
o := orm.NewOrm()
user := User{Id: 1}
if o.Read(&user) == nil {
    user.Name = "MyName"
    if num, err := o.Update(&user); err == nil {
        fmt.Println(num)
    }
}
```

Update 默认更新所有的字段，可以更新指定的字段：

```
// 只更新 Name
o.Update(&user, "Name")
// 指定多个字段
// o.Update(&user, "Field1", "Field2", ...)
...
```

根据复杂条件更新字段值参见 Update

Delete

第一个返回值为影响的行数

```
o := orm.NewOrm()
if num, err := o.Delete(&User{Id: 1}); err == nil {
    fmt.Println(num)
}
```

Delete 操作会对反向关系进行操作，此例中 Post 拥有一个到 User 的外键。删除 User 的时候。如果 on_delete 设置为默认的级联操作，将删除对应的 Post

Changed in 1.0.3 删除以后不会删除 auto field 的值

高级查询

高级查询

ORM 以 **QuerySeter** 来组织查询，每个返回 **QuerySeter** 的方法都会获得一个新的 **QuerySeter** 对象。

基本使用方法:

```
o := orm.NewOrm()

// 获取 QuerySeter 对象，user 为表名
qs := o.QueryTable("user")

// 也可以直接使用对象作为表名
user := new(User)
qs = o.QueryTable(user) // 返回 QuerySeter
```

expr

QuerySeter 中用于描述字段和 sql 操作符，使用简单的 expr 查询方法

字段组合的前后顺序依照表的关系，比如 User 表拥有 Profile 的外键，那么对 User 表查询对应的 Profile.Age 为条件，则使用 `Profile__Age` 注意，字段的分隔符号使用双下划线 `__` ，除了描述字段， expr 的尾部可以增加操作符以执行对应的 sql 操作。比如 `Profile__Age__gt` 代表 Profile.Age > 18 的条件查询。

注释后面将描述对应的 sql 语句，仅仅是描述 expr 的类似结果，并不代表实际生成的语句。

```
qs.Filter("id", 1) // WHERE id = 1
qs.Filter("profile__age", 18) // WHERE profile.age = 18
qs.Filter("Profile__Age", 18) // 使用字段名和 Field 名都是允许的
qs.Filter("profile__age", 18) // WHERE profile.age = 18
qs.Filter("profile__age__gt", 18) // WHERE profile.age > 18
qs.Filter("profile__age__gte", 18) // WHERE profile.age >= 18
qs.Filter("profile__age__in", 18, 20) // WHERE profile.age IN (18, 20)

qs.Filter("profile__age__in", 18, 20).Exclude("profile__lt", 1000)
// WHERE profile.age IN (18, 20) AND NOT profile_id < 1000
```

Operators

当前支持的操作符号：

- exact / iexact 等于
- contains / icontains 包含
- [gt / gte](#gt / gte) 大于 / 大于等于
- [lt / lte](#lt / lte) 小于 / 小于等于
- startswith / istartswith 以...起始
- endswith / iendswith 以...结束
- in
- isnull

后面以 `i` 开头的表示：大小写不敏感

exact

Filter / Exclude / Condition expr 的默认值

```
qs.Filter("name", "slene") // WHERE name = 'slene'
qs.Filter("name__exact", "slene") // WHERE name = 'slene'
// 使用 = 匹配，大小写是否敏感取决于数据表使用的 collation
qs.Filter("profile_id", nil) // WHERE profile_id IS NULL
```

iexact

```
qs.Filter("name__iexact", "slene")
// WHERE name LIKE 'slene'
// 大小写不敏感，匹配任意 'Slene' 'sLENE'
```

contains

```
qs.Filter("name__contains", "slene")
// WHERE name LIKE BINARY '%slene%'
// 大小写敏感，匹配包含 slene 的字符
```

icontains

```
qs.Filter("name__icontains", "slene")
// WHERE name LIKE '%slene%'
// 大小写不敏感，匹配任意 'im Slene', 'im sLENE'
```

in

```
qs.Filter("age__in", 17, 18, 19, 20)
// WHERE age IN (17, 18, 19, 20)


ids:=[]int{17,18,19,20}
qs.Filter("age__in", ids)
// WHERE age IN (17, 18, 19, 20)

// 同上效果
```

gt / gte

```
qs.Filter("profile__age__gt", 17)
// WHERE profile.age > 17

qs.Filter("profile__age__gte", 18)
// WHERE profile.age >= 18
```

lt / lte

```
qs.Filter("profile__age__lt", 17)
// WHERE profile.age < 17

qs.Filter("profile__age__lte", 18)
// WHERE profile.age <= 18
```

startswith

```
qs.Filter("name__startswith", "slene")
// WHERE name LIKE BINARY 'slene%'
// 大小写敏感，匹配以 'slene' 起始的字符串
```

istartswith

```
qs.Filter("name__istartswith", "slene")
// WHERE name LIKE 'slene%'
// 大小写不敏感，匹配任意以 'slene', 'Slene' 起始的字符串
```

endswith

```
qs.Filter("name__endswith", "slene")
// WHERE name LIKE BINARY '%slene'
// 大小写敏感，匹配以 'slene' 结束的字符串
```

iendswith

```
qs.Filter("name__iendswithi", "slene")
// WHERE name LIKE '%slene'
// 大小写不敏感，匹配任意以 'slene', 'Slene' 结束的字符串
```

isnull

```
qs.Filter("profile__isnull", true)
qs.Filter("profile_id__isnull", true)
// WHERE profile_id IS NULL

qs.Filter("profile__isnull", false)
// WHERE profile_id IS NOT NULL
```

高级查询接口使用

QuerySeter 是高级查询使用的接口，我们来熟悉下他的接口方法

- type QuerySeter interface {
    - Filter(string, ...interface{}) QuerySeter
    - Exclude(string, ...interface{}) QuerySeter
    - SetCond(*Condition) QuerySeter
    - Limit(int, ...int64) QuerySeter
    - Offset(int64) QuerySeter
    - GroupBy(...string) QuerySeter
    - OrderBy(...string) QuerySeter

    - Distinct() QuerySeter
    - RelatedSel(...interface{}) QuerySeter
    - Count() (int64, error)
    - Exist() bool
    - Update(Params) (int64, error)
    - Delete() (int64, error)
    - PrepareInsert() (Inserter, error)
    - All(interface{}, ...string) (int64, error)
    - One(interface{}, ...string) error
    - Values(*[]Params, ...string) (int64, error)
    - ValuesList(*[]ParamsList, ...string) (int64, error)
    - ValuesFlat(*ParamsList, string) (int64, error)
- }
- 每个返回 QuerySeter 的 api 调用时都会新建一个 QuerySeter，不影响之前创建的。
- 高级查询使用 Filter 和 Exclude 来做常用的条件查询。囊括两种清晰的过滤规则：包含，排除

Filter

用来过滤查询结果，起到 **包含条件** 的作用

多个 Filter 之间使用 `AND` 连接

```
qs.Filter("profile__isnull", true).Filter("name", "slene")
// WHERE profile_id IS NULL AND name = 'slene'
```

Exclude

用来过滤查询结果，起到 **排除条件** 的作用

使用 `NOT` 排除条件

多个 Exclude 之间使用 `AND` 连接

```
qs.Exclude("profile__isnull", true).Filter("name", "slene")
// WHERE NOT profile_id IS NULL AND name = 'slene'
```

SetCond

自定义条件表达式

```
cond := orm.NewCondition()
cond1 := cond.And("profile__isnull", false).AndNot("status__in", 1).Or("profile__age__gt", 2000)

qs := orm.QueryTable("user")
qs = qs.SetCond(cond1)
// WHERE ... AND ... AND NOT ... OR ...

cond2 := cond.AndCond(cond1).OrCond(cond.And("name", "slene"))
qs = qs.SetCond(cond2).Count()
// WHERE (... AND ... AND NOT ... OR ...) OR ( ... )
```

Limit

限制最大返回数据行数，第二个参数可以设置 `Offset`

```
var DefaultRowsLimit = 1000 // ORM 默认的 limit 值为 1000

// 默认情况下 select 查询的最大行数为 1000
// LIMIT 1000

qs.Limit(10)
// LIMIT 10

qs.Limit(10, 20)
// LIMIT 10 OFFSET 20 注意跟 SQL 反过来的

qs.Limit(-1)
// no limit

qs.Limit(-1, 100)
// LIMIT 18446744073709551615 OFFSET 100
// 18446744073709551615 是 1<<64 - 1 用来指定无 limit 限制 但有 offset 偏移的情况
```

Offset

设置 偏移行数

```
qs.Offset(20)
// LIMIT 1000 OFFSET 20
```

GroupBy

```
qs.GroupBy("id", "age")
// GROUP BY id,age
```

OrderBy

参数使用 **expr**

在 expr 前使用减号 `-` 表示 `DESC` 的排列

```
qs.OrderBy("id", "-profile__age")
// ORDER BY id ASC, profile.age DESC

qs.OrderBy("-profile__age", "profile")
// ORDER BY profile.age DESC, profile_id ASC
```

Distinct

对应 sql 的 `distinct` 语句, 返回不重复的值.

```
qs.Distinct()
// SELECT DISTINCT
```

RelatedSel

关系查询，参数使用 **expr**

```
var DefaultRelsDepth = 5 // 默认情况下直接调用 RelatedSel 将进行最大 5 层的关系查询

qs := o.QueryTable("post")

qs.RelatedSel()
// INNER JOIN user ... LEFT OUTER JOIN profile ...

qs.RelatedSel("user")
// INNER JOIN user ...
```

```
// 设置 expr 只对设置的字段进行关系查询

// 对设置 null 属性的 Field 将使用 LEFT OUTER JOIN
```

Count

依据当前的查询条件，返回结果行数

```
cnt, err := o.QueryTable("user").Count() // SELECT COUNT(*) FROM USER
fmt.Printf("Count Num: %s, %s", cnt, err)
```

Exist

```
exist := o.QueryTable("user").Filter("UserName", "Name").Exist()
fmt.Printf("Is Exist: %s", exist)
```

Update

依据当前查询条件，进行批量更新操作

```
num, err := o.QueryTable("user").Filter("name", "slene").Update(orm.Params{
    "name": "astaxie",
})
fmt.Printf("Affected Num: %s, %s", num, err)
// SET name = "astaixe" WHERE name = "slene"
```

原子操作增加字段值

```
// 假设 user struct 里有一个 nums int 字段
num, err := o.QueryTable("user").Update(orm.Params{
    "nums": orm.ColValue(orm.ColAdd, 100),
})
// SET nums = nums + 100
```

orm.ColValue 支持以下操作

```
ColAdd      // 加
ColMinus    // 减
ColMultiply // 乘
ColExcept   // 除
```

Delete

依据当前查询条件，进行批量删除操作

```
num, err := o.QueryTable("user").Filter("name", "slene").Delete()
fmt.Printf("Affected Num: %s, %s", num, err)
// DELETE FROM user WHERE name = "slene"
```

PrepareInsert

用于一次 prepare 多次 insert 插入，以提高批量插入的速度。

```
var users []*User
...
qs := o.QueryTable("user")
i, _ := qs.PrepareInsert()
for _, user := range users {
    id, err := i.Insert(user)
    if err == nil {
        ...
    }
}
// PREPARE INSERT INTO user (`name`, ...) VALUES (?, ...)
// EXECUTE INSERT INTO user (`name`, ...) VALUES ("slene", ...)
// EXECUTE ...
// ...
i.Close() // 别忘记关闭 statement
```

All

返回对应的结果集对象

All 的参数支持 *[]Type 和 *[]*Type 两种形式的 slice

```
var users []*User
num, err := o.QueryTable("user").Filter("name", "slene").All(&users)
fmt.Printf("Returned Rows Num: %s, %s", num, err)
```

All / Values / ValuesList / ValuesFlat 受到 Limit 的限制，默认最大行数为 1000

可以指定返回的字段：

```
type Post struct {
    Id      int
    Title   string
    Content string
    Status  int
}

// 只返回 Id 和 Title
var posts []Post
o.QueryTable("post").Filter("Status", 1).All(&posts, "Id", "Title")
```

对象的其他字段值将会是对应类型的默认值

One

尝试返回单条记录

```
var user User
err := o.QueryTable("user").Filter("name", "slene").One(&user)
if err == orm.ErrMultiRows {
    // 多条的时候报错
    fmt.Printf("Returned Multi Rows Not One")
}
if err == orm.ErrNoRows {
    // 没有找到记录
    fmt.Printf("Not row found")
}
```

可以指定返回的字段：

```
// 只返回 Id 和 Title
var post Post
o.QueryTable("post").Filter("Content__istartswith", "prefix string").One(&post,
"Id", "Title")
```

对象的其他字段值将会是对应类型的默认值

Values

返回结果集的 key => value 值

key 为Model里的Field name, value的值是interface{}类型,例如，如果你要将value赋值给struct中的某字段，需要根据结构体对应字段类型使用断言获取真实值。举例：`Name :`

`m["Name"].(string)`

```
var maps []orm.Params
num, err := o.QueryTable("user").Values(&maps)
if err == nil {
    fmt.Printf("Result Nums: %d\n", num)
    for _, m := range maps {
        fmt.Println(m["Id"], m["Name"])
    }
}
```

返回指定的 Field 数据

TODO: 暂不支持级联查询 **RelatedSel** 直接返回 Values

但可以直接指定 expr 级联返回需要的数据

```
var maps []orm.Params
num, err := o.QueryTable("user").Values(&maps, "id", "name", "profile", "profile__age")
if err == nil {
    fmt.Printf("Result Nums: %d\n", num)
    for _, m := range maps {
        fmt.Println(m["Id"], m["Name"], m["Profile"], m["Profile__Age"])
        // map 中的数据都是展开的，没有复杂的嵌套
    }
}
```

ValuesList

顾名思义，返回的结果集以slice存储

结果的排列与 Model 中定义的 Field 顺序一致

返回的每个元素值以 string 保存

```
var lists []orm.ParamsList
num, err := o.QueryTable("user").ValuesList(&lists)
if err == nil {
    fmt.Printf("Result Nums: %d\n", num)
    for _, row := range lists {
        fmt.Println(row)
    }
}
```

当然也可以指定 expr 返回指定的 Field

```
var lists []orm.ParamsList
num, err := o.QueryTable("user").ValuesList(&lists, "name", "profile__age")
if err == nil {
    fmt.Printf("Result Nums: %d\n", num)
    for _, row := range lists {
        fmt.Printf("Name: %s, Age: %s\m", row[0], row[1])
    }
}
```

ValuesFlat

只返回特定的 Field 值，将结果集展开到单个 slice 里

```
var list orm.ParamsList
num, err := o.QueryTable("user").ValuesFlat(&list, "name")
if err == nil {
    fmt.Printf("Result Nums: %d\n", num)
    fmt.Printf("All User Names: %s", strings.Join(list, ", "))
}
```

关系查询

以例子里的模型定义来看下怎么进行关系查询

User 和 Profile 是 OneToOne 的关系

已经取得了 User 对象，查询 Profile：

```
user := &User{Id: 1}
o.Read(user)
if user.Profile != nil {
    o.Read(user.Profile)
}
```

直接关联查询：

```
user := &User{}
o.QueryTable("user").Filter("Id", 1).RelatedSel().One(user)
// 自动查询到 Profile
fmt.Println(user.Profile)
```

```
// 因为在 Profile 里定义了反向关系的 User，所以 Profile 里的 User 也是自动赋值过的，可以直接取用。
fmt.Println(user.Profile.User)

// [SELECT T0.`id`, T0.`name`, T0.`profile_id`, T1.`id`, T1.`age` FROM `user` T0 INNER JOIN `profile` T1 ON T1.`id` = T0.`profile_id` WHERE T0.`id` = ? LIMIT 1000] - `1`
```

通过 User 反向查询 Profile：

```
var profile Profile
err := o.QueryTable("profile").Filter("User__Id", 1).One(&profile)
if err == nil {
    fmt.Println(profile)
}
```

Post 和 User 是 ManyToOne 关系，也就是 ForeignKey 为 User

```
type Post struct {
    Id    int
    Title string
    User  *User  `orm:"rel(fk)"`
    Tags  []*Tag `orm:"rel(m2m)"`
}
```

```
var posts []*Post
num, err := o.QueryTable("post").Filter("User", 1).RelatedSel().All(&posts)
if err == nil {
    fmt.Printf("%d posts read\n", num)
    for _, post := range posts {
        fmt.Printf("Id: %d, UserName: %d, Title: %s\n", post.Id, post.User.UserName, post.Title)
    }
}
// [SELECT T0.`id`, T0.`title`, T0.`user_id`, T1.`id`, T1.`name`, T1.`profile_id`, T2.`id`, T2.`age` FROM `post` T0 INNER JOIN `user` T1 ON T1.`id` = T0.`user_id` INNER JOIN `profile` T2 ON T2.`id` = T1.`profile_id` WHERE T0.`user_id` = ? LIMIT 1000] - `1`
```

根据 Post.Title 查询对应的 User：

RegisterModel 时，ORM 也会自动建立 User 中 Post 的反向关系，所以可以直接进行查询

```
var user User
err := o.QueryTable("user").Filter("Post__Title", "The Title").Limit(1).One(&use
r)
if err == nil {
    fmt.Printf(user)
}
```

Post 和 Tag 是 ManyToMany 关系

设置 rel(m2m) 以后，ORM 会自动创建中间表

```
type Post struct {
    Id    int
    Title string
    User  *User  `orm:"rel(fk)"`
    Tags  []*Tag `orm:"rel(m2m)"`
}
```

```
type Tag struct {
    Id    int
    Name  string
    Posts []*Post `orm:"reverse(many)"`
}
```

一条 Post 纪录可能对应不同的 Tag 纪录,一条 Tag 纪录可能对应不同的 Post 纪录，所以 Post 和 Tag 属于多对多关系,通过 tag name 查询哪些 post 使用了这个 tag

```
var posts []*Post
num, err := dORM.QueryTable("post").Filter("Tags__Tag__Name", "golang").All(&pos
ts)
```

通过 post title 查询这个 post 有哪些 tag

```
var tags []*Tag
num, err := dORM.QueryTable("tag").Filter("Posts__Post__Title", "Introduce Beego
ORM").All(&tags)
```

载入关系字段

LoadRelated 用于载入模型的关系字段，包括所有的 rel/reverse - one/many 关系

ManyToMany 关系字段载入

```
// 载入相应的 Tags
post := Post{Id: 1}
err := o.Read(&post)
num, err := o.LoadRelated(&post, "Tags")
```

```
// 载入相应的 Posts
tag := Tag{Id: 1}
err := o.Read(&tag)
num, err := o.LoadRelated(&tag, "Posts")
```

User 是 Post 的 ForeignKey，对应的 ReverseMany 关系字段载入

```
type User struct {
    Id    int
    Name  string
    Posts []*Post `orm:"reverse(many)"`
}

user := User{Id: 1}
err := dORM.Read(&user)
num, err := dORM.LoadRelated(&user, "Posts")
for _, post := range user.Posts {
    //...
}
```

多对多关系操作

- type QueryM2Mer interface {
  - Add(...interface{}) (int64, error)
  - Remove(...interface{}) (int64, error)
  - Exist(interface{}) bool
  - Clear() (int64, error)
  - Count() (int64, error)
- }

创建一个 QueryM2Mer 对象

```
o := orm.NewOrm()
post := Post{Id: 1}
m2m := o.QueryM2M(&post, "Tags")
// 第一个参数的对象，主键必须有值
// 第二个参数为对象需要操作的 M2M 字段
// QueryM2Mer 的 api 将作用于 Id 为 1 的 Post
```

QueryM2Mer Add

```
tag := &Tag{Name: "golang"}
o.Insert(tag)

num, err := m2m.Add(tag)
if err == nil {
    fmt.Println("Added nums: ", num)
}
```

Add 支持多种类型 Tag *Tag* *[]*Tag []Tag []interface{}

```
var tags []*Tag
...
// 读取 tags 以后
...
num, err := m2m.Add(tags)
if err == nil {
    fmt.Println("Added nums: ", num)
}
// 也可以多个作为参数传入
// m2m.Add(tag1, tag2, tag3)
```

QueryM2Mer Remove

从M2M关系中删除 tag

Remove 支持多种类型 Tag *Tag* *[]*Tag []Tag []interface{}

```
var tags []*Tag
...
// 读取 tags 以后
...
num, err := m2m.Remove(tags)
if err == nil {
    fmt.Println("Removed nums: ", num)
```

```
}
// 也可以多个作为参数传入
// m2m.Remove(tag1, tag2, tag3)
```

QueryM2Mer Exist

判断 Tag 是否存在于 M2M 关系中

```
if m2m.Exist(&Tag{Id: 2}) {
    fmt.Println("Tag Exist")
}
```

QueryM2Mer Clear

清除所有 M2M 关系

```
nums, err := m2m.Clear()
if err == nil {
    fmt.Println("Removed Tag Nums: ", nums)
}
```

QueryM2Mer Count

计算 Tag 的数量

```
nums, err := m2m.Count()
if err == nil {
    fmt.Println("Total Nums: ", nums)
}
```

原生SQL查询

原生SQL查询

使用SQL语句进行查询

- 使用 Raw SQL 查询，无需使用 ORM 表定义
- 多数据库，都可直接使用占位符号 `?` ，自动转换
- 查询时的参数，支持使用 Model Struct 和 Slice, Array

```
ids := []int{1, 2, 3}
p.Raw("SELECT name FROM user WHERE id IN (?, ?, ?)", ids)
```

创建一个 **RawSeter**

```
o := orm.NewOrm()
var r RawSeter
r = o.Raw("UPDATE user SET name = ? WHERE name = ?", "testing", "slene")
```

- type RawSeter interface {
  - Exec() (sql.Result, error)
  - QueryRow(...interface{}) error
  - QueryRows(...interface{}) (int64, error)
  - SetArgs(...interface{}) RawSeter
  - Values(*[]Params, ...string) (int64, error)
  - ValuesList(*[]ParamsList, ...string) (int64, error)
  - ValuesFlat(*ParamsList, string) (int64, error)
  - RowsToMap(*Params, string, string) (int64, error)
  - RowsToStruct(interface{}, string, string) (int64, error)
  - Prepare() (RawPreparer, error)
- }

Exec

执行 sql 语句，返回 sql.Result 对象

```
res, err := o.Raw("UPDATE user SET name = ?", "your").Exec()
if err == nil {
    num, _ := res.RowsAffected()
    fmt.Println("mysql row affected nums: ", num)
}
```

QueryRow

QueryRow 和 QueryRows 提供高级 sql mapper 功能

支持 struct

```
type User struct {
    Id       int
    UserName string
}

var user User
err := o.Raw("SELECT id, user_name FROM user WHERE id = ?", 1).QueryRow(&user)
```

> from beego 1.1.0 取消了多个对象支持 ISSUE 384

QueryRows

QueryRows 支持的对象还有 map 规则是和 QueryRow 一样的，但都是 slice

```
type User struct {
    Id       int
    UserName string
}

var users []User
num, err := o.Raw("SELECT id, user_name FROM user WHERE id = ?", 1).QueryRows(&u
sers)
if err == nil {
    fmt.Println("user nums: ", num)
}
```

> from beego 1.1.0 取消了多个对象支持 ISSUE 384
> `

SetArgs

改变 Raw(sql, args...) 中的 args 参数，返回一个新的 RawSeter

用于单条 sql 语句，重复利用，替换参数然后执行。

```
res, err := r.SetArgs("arg1", "arg2").Exec()
res, err := r.SetArgs("arg1", "arg2").Exec()
...
```

Values / ValuesList / ValuesFlat

Raw SQL 查询获得的结果集 Value 为 `string` 类型，NULL 字段的值为空 ``

> from beego 1.1.0
> Values, ValuesList, ValuesFlat 的参数，可以指定返回哪些 Columns 的数据
> 通常情况下，是无需指定的，因为 sql 语句中你可以自行设置 SELECT 的字段

Values

返回结果集的 key => value 值

```
var maps []orm.Params
num, err := o.Raw("SELECT user_name FROM user WHERE status = ?", 1).Values(&maps)
if err == nil && num > 0 {
    fmt.Println(maps[0]["user_name"]) // slene
}
```

ValuesList

返回结果集 slice

```
var lists []orm.ParamsList
num, err := o.Raw("SELECT user_name FROM user WHERE status = ?", 1).ValuesList(&lists)
if err == nil && num > 0 {
    fmt.Println(lists[0][0]) // slene
}
```

ValuesFlat

返回单一字段的平铺 slice 数据

```
var list orm.ParamsList
num, err := o.Raw("SELECT id FROM user WHERE id < ?", 10).ValuesFlat(&list)
if err == nil && num > 0 {
    fmt.Println(list) // []{"1","2","3",...}
}
```

RowsToMap

SQL 查询结果是这样

name	value
total	100
found	200

查询结果匹配到 map 里

```
res := make(orm.Params)
nums, err := o.Raw("SELECT name, value FROM options_table").RowsToMap(&res, "name", "value")
// res is a map[string]interface{}{
//     "total": 100,
//     "found": 200,
// }
```

RowsToStruct

SQL 查询结果是这样

name	value
total	100
found	200

查询结果匹配到 struct 里

```
type Options struct {
    Total int
    Found int
}
```

```
res := new(Options)
nums, err := o.Raw("SELECT name, value FROM options_table").RowsToStruct(res, "name", "value")
fmt.Println(res.Total) // 100
fmt.Println(res.Found) // 200
```

> 匹配支持的名称转换为 snake -> camel, eg: SELECT user_name ... 需要你的 struct 中定义有 UserName

Prepare

用于一次 prepare 多次 exec，以提高批量执行的速度。

```
p, err := o.Raw("UPDATE user SET name = ? WHERE name = ?").Prepare()
res, err := p.Exec("testing", "slene")
res, err  = p.Exec("testing", "astaxie")
...
...
p.Close() // 别忘记关闭 statement
```

构造查询

构造查询

QueryBuilder 提供了一个简便，流畅的 SQL 查询构造器。在不影响代码可读性的前提下用来快速的建立 SQL 语句。

QueryBuilder 在功能上与 ORM 重合， 但是各有利弊。ORM 更适用于简单的 CRUD 操作，而 QueryBuilder 则更适用于复杂的查询，例如查询中包含子查询和多重联结。

使用方法:

```
// User 包装了下面的查询结果
type User struct {
    Name string
    Age  int
}
var users []User

// 获取 QueryBuilder 对象. 需要指定数据库驱动参数。
// 第二个返回值是错误对象，在这里略过
qb, _ := orm.NewQueryBuilder("mysql")

// 构建查询对象
qb.Select("user.name",
    "profile.age").
    From("user").
    InnerJoin("profile").On("user.id_user = profile.fk_user").
    Where("age > ?").
    OrderBy("name").Desc().
    Limit(10).Offset(0)

// 导出 SQL 语句
sql := qb.String()

// 执行 SQL 语句
o := orm.NewOrm()
o.Raw(sql, 20).QueryRows(&users)
```

完整 API 接口:

```
type QueryBuilder interface {
    Select(fields ...string) QueryBuilder
    From(tables ...string) QueryBuilder
    InnerJoin(table string) QueryBuilder
    LeftJoin(table string) QueryBuilder
    RightJoin(table string) QueryBuilder
    On(cond string) QueryBuilder
    Where(cond string) QueryBuilder
    And(cond string) QueryBuilder
    Or(cond string) QueryBuilder
    In(vals ...string) QueryBuilder
    OrderBy(fields ...string) QueryBuilder
    Asc() QueryBuilder
    Desc() QueryBuilder
    Limit(limit int) QueryBuilder
    Offset(offset int) QueryBuilder
    GroupBy(fields ...string) QueryBuilder
    Having(cond string) QueryBuilder
    Subquery(sub string, alias string) string
    String() string
}
```

事务处理

事务处理

ORM 可以简单的进行事务操作

```
o := NewOrm()
err := o.Begin()
// 事务处理过程
...
...
// 此过程中的所有使用 o Ormer 对象的查询都在事务处理范围内
if SomeError {
    err = o.Rollback()
} else {
    err = o.Commit()
}
```

模型定义

模型定义

复杂的模型定义不是必须的，此功能用作数据库数据转换和自动建表

默认的表名规则，使用驼峰转蛇形：

```
AuthUser -> auth_user
Auth_User -> auth__user
DB_AuthUser -> d_b__auth_user
```

除了开头的大写字母以外，遇到大写会增加 `_` ，原名称中的下划线保留。

自定义表名

```
type User struct {
    Id int
    Name string
}

func (u *User) TableName() string {
    return "auth_user"
}
```

如果前缀设置为 `prefix_` 那么表名为：prefix_auth_user

自定义索引

为单个或多个字段增加索引

```
type User struct {
    Id    int
    Name  string
    Email string
}

// 多字段索引
func (u *User) TableIndex() [][]string {
```

```
    return [][]string{
        []string{"Id", "Name"},
    }
}

// 多字段唯一键
func (u *User) TableUnique() [][]string {
    return [][]string{
        []string{"Name", "Email"},
    }
}
```

自定义引擎

仅支持 MySQL

默认使用的引擎，为当前数据库的默认引擎，这个是由你的 mysql 配置参数决定的。

你可以在模型里设置 TableEngine 函数，指定使用的引擎

```
type User struct {
    Id    int
    Name  string
    Email string
}

// 设置引擎为 INNODB
func (u *User) TableEngine() string {
    return "INNODB"
}
```

设置参数

```
orm:"null;rel(fk)"
```

多个设置间使用 `;` 分隔，设置的值如果是多个，使用 `,` 分隔。

忽略字段

设置 `-` 即可忽略 struct 中的字段

```
type User struct {
...
    AnyField string `orm:"-"`
...
}
```

auto

当 Field 类型为 int, int32, int64, uint, uint32, uint64 时，可以设置字段为自增健

- 当模型定义里没有主键时，符合上述类型且名称为 `Id` 的 Field 将被视为自增健。

鉴于 go 目前的设计，即使使用了 uint64，但你也不能存储到他的最大值。依然会作为 int64 处理。

参见 issue 6113

pk

设置为主键，适用于自定义其他类型为主键

null

数据库表默认为 `NOT NULL` ，设置 null 代表 `ALLOW NULL`

```
Name string `orm:"null"`
```

index

为单个字段增加索引

unique

为单个字段增加 unique 键

```
Name string `orm:"unique"`
```

column

为字段设置 db 字段的名称

```
Name string `orm:"column(user_name)"`
```

size

string 类型字段默认为 varchar(255)

设置 size 以后，db type 将使用 varchar(size)

```
Title string `orm:"size(60)"`
```

digits / decimals

设置 float32, float64 类型的浮点精度

```
Money float64 `orm:"digits(12);decimals(4)"`
```

总长度 12 小数点后 4 位 eg: `99999999.9999`

auto_now / auto_now_add

```
Created time.Time `orm:"auto_now_add;type(datetime)"`
Updated time.Time `orm:"auto_now;type(datetime)"`
```

- auto_now 每次 model 保存时都会对时间自动更新
- auto_now_add 第一次保存时才设置时间

对于批量的 update 此设置是不生效的

type

设置为 date 时，time.Time 字段的对应 db 类型使用 date

```
Created time.Time `orm:"auto_now_add;type(date)"`
```

设置为 datetime 时，time.Time 字段的对应 db 类型使用 datetime

```
Created time.Time `orm:"auto_now_add;type(datetime)"`
```

default

为字段设置默认值，类型必须符合（目前仅用于级联删除时的默认值）

```
type User struct {
    ...
    Status int `orm:"default(1)"`
    ...
}
```

Comment

为字段添加注释

```
type User struct {
    ...
    Status int `orm:"default(1)" description:"这是状态字段"`
    ...
}
```

注意: 注释中禁止包含引号

表关系设置

rel / reverse

RelOneToOne:

```
type User struct {
    ...
    Profile *Profile `orm:"null;rel(one);on_delete(set_null)"`
    ...
}
```

对应的反向关系 **RelReverseOne**:

```
type Profile struct {
    ...
    User *User `orm:"reverse(one)"`
    ...
}
```

RelForeignKey:

```
type Post struct {
    ...
```

```
    User *User `orm:"rel(fk)"` // RelForeignKey relation
    ...
}
```

对应的反向关系 **RelReverseMany**:

```
type User struct {
    ...
    Posts []*Post `orm:"reverse(many)"` // fk 的反向关系
    ...
}
```

RelManyToMany:

```
type Post struct {
    ...
    Tags []*Tag `orm:"rel(m2m)"` // ManyToMany relation
    ...
}
```

对应的反向关系 **RelReverseMany**:

```
type Tag struct {
    ...
    Posts []*Post `orm:"reverse(many)"`
    ...
}
```

rel_table / rel_through

此设置针对 `orm:"rel(m2m)"` 的关系字段

```
rel_table       设置自动生成的 m2m 关系表的名称
rel_through     如果要在 m2m 关系中使用自定义的 m2m 关系表
                通过这个设置其名称，格式为 pkg.path.ModelName
                eg: app.models.PostTagRel
                PostTagRel 表需要有到 Post 和 Tag 的关系
```

当设置 rel_table 时会忽略 rel_through

设置方法：

`orm:"rel(m2m);rel_table(the_table_name)"`

```
orm:"rel(m2m);rel_through(pkg.path.ModelName)"
```

on_delete

设置对应的 rel 关系删除时，如何处理关系字段。

```
cascade         级联删除(默认值)
set_null        设置为 NULL，需要设置 null = true
set_default     设置为默认值，需要设置 default 值
do_nothing      什么也不做，忽略
```

```
type User struct {
    ...
    Profile *Profile `orm:"null;rel(one);on_delete(set_null)"`
    ...
}
type Profile struct {
    ...
    User *User `orm:"reverse(one)"`
    ...
}

// 删除 Profile 时将设置 User.Profile 的数据库字段为 NULL
```

关于 on_delete 的相关例子

```
type User struct {
    Id int
    Name string
}

type Post struct {
    Id int
    Title string
    User *User `orm:"rel(fk)"`
}
```

假设 Post -> User 是 ManyToOne 的关系，也就是外键。

```
o.Filter("Id", 1).Delete()
```

这个时候即会删除 Id 为 1 的 User 也会删除其发布的 Post

不想删除的话，需要设置 set_null

```
type Post struct {
    Id int
    Title string
    User *User `orm:"rel(fk);null;on_delete(set_null)"`
}
```

那这个时候，删除 User 只会把对应的 Post.user_id 设置为 NULL

当然有时候为了高性能的需要，多存点数据无所谓啊，造成批量删除才是问题。

```
type Post struct {
    Id int
    Title string
    User *User `orm:"rel(fk);null;on_delete(do_nothing)"`
}
```

那么只要删除的时候，不操作 Post 就可以了。

模型字段与数据库类型的对应

在此列出 ORM 推荐的对应数据库类型，自动建表功能也会以此为标准。

默认所有的字段都是 **NOT NULL**

MySQL

go	mysql
int, int32 - 设置 auto 或者名称为 `Id` 时	integer AUTO_INCREMENT
int64 - 设置 auto 或者名称为 `Id` 时	bigint AUTO_INCREMENT
uint, uint32 - 设置 auto 或者名称为 `Id` 时	integer unsigned AUTO_INCREMENT
uint64 - 设置 auto 或者名称为 `Id` 时	bigint unsigned AUTO_INCREMENT
bool	bool
string - 默认为 size 255	varchar(size)
string - 设置 type(char) 时	char(size)

go	mysql
string - 设置 type(text) 时	longtext
time.Time - 设置 type 为 date 时	date
time.Time	datetime
byte	tinyint unsigned
rune	integer
int	integer
int8	tinyint
int16	smallint
int32	integer
int64	bigint
uint	integer unsigned
uint8	tinyint unsigned
uint16	smallint unsigned
uint32	integer unsigned
uint64	bigint unsigned
float32	double precision
float64	double precision
float64 - 设置 digits, decimals 时	numeric(digits, decimals)

Sqlite3

go	sqlite3
int, int32, int64, uint, uint32, uint64 - 设置 auto 或者名称为 `Id` 时	integer AUTOINCREMENT
bool	bool

go	sqlite3
string - 默认为 size 255	varchar(size)
string - 设置 type(char) 时	character(size)
string - 设置 type(text) 时	text
time.Time - 设置 type 为 date 时	date
time.Time	datetime
byte	tinyint unsigned
rune	integer
int	integer
int8	tinyint
int16	smallint
int32	integer
int64	bigint
uint	integer unsigned
uint8	tinyint unsigned
uint16	smallint unsigned
uint32	integer unsigned
uint64	bigint unsigned
float32	real
float64	real
float64 - 设置 digits, decimals 时	decimal

PostgreSQL

go	postgres

go	postgres
int, int32, int64, uint, uint32, uint64 - 设置 auto 或者名称为 `Id` 时	serial
bool	bool
string - 若没有指定 size 默认为 text	varchar(size)
string - 设置 type(char) 时	char(size)
string - 设置 type(text) 时	text
string - 设置 type(json) 时	json
string - 设置 type(jsonb) 时	jsonb
time.Time - 设置 type 为 date 时	date
time.Time	timestamp with time zone
byte	smallint CHECK(“column” >= 0 AND “column” <= 255)
rune	integer
int	integer
int8	smallint CHECK(“column” >= -127 AND “column” <= 128)
int16	smallint
int32	integer
int64	bigint
uint	bigint CHECK(“column” >= 0)
uint8	smallint CHECK(“column” >= 0 AND “column” <= 255)
uint16	integer CHECK(“column” >= 0)
uint32	bigint CHECK(“column” >= 0)
uint64	bigint CHECK(“column” >= 0)

go	postgres
float32	double precision
float64	double precision
float64 - 设置 digits, decimals 时	numeric(digits, decimals)

关系型字段

其字段类型取决于对应的主键。

- RelForeignKey
- RelOneToOne
- RelManyToMany
- RelReverseOne
- RelReverseMany

命令模式

命令模式

注册模型与数据库以后，调用 RunCommand 执行 orm 命令。

```
func main() {
    // orm.RegisterModel...
    // orm.RegisterDataBase...
    ...
    orm.RunCommand()
}
```

```
go build main.go
./main orm
# 直接执行可以显示帮助
# 如果你的程序可以支持的话，直接运行 go run main.go orm 也是一样的效果
```

测试用例

测试用例

测试代码参见

- 表定义 models_test.go
- 测试用例 orm_test.go

MySQL

```
mysql -u root -e 'create database orm_test;'
export ORM_DRIVER=mysql
export ORM_SOURCE="root:@/orm_test?charset=utf8"
go test -v github.com/astaxie/beego/orm
```

Sqlite3

```
touch /path/to/orm_test.db
export ORM_DRIVER=sqlite3
export ORM_SOURCE=/path/to/orm_test.db
go test -v github.com/astaxie/beego/orm
```

PostgreSQL

```
psql -c 'create database orm_test;' -U postgres
export ORM_DRIVER=postgres
export ORM_SOURCE="user=postgres dbname=orm_test sslmode=disable"
go test -v github.com/astaxie/beego/orm
```

自定义字段

自定义字段

Custom Fields

```
TypeBooleanField = 1 << iota

// string
TypeCharField

// string
TypeTextField

// time.Time
TypeDateField
// time.Time
TypeDateTimeField

// int16
TypeSmallIntegerField
// int32
TypeIntegerField
// int64
TypeBigIntegerField
// uint16
TypePositiveSmallIntegerField
// uint32
TypePositiveIntegerField
// uint64
TypePositiveBigIntegerField

// float64
TypeFloatField
// float64
TypeDecimalField

RelForeignKey
RelOneToOne
RelManyToMany
```

```
RelReverseOne
RelReverseMany
```

FAQ

FAQ

1. 我的 app 需要支持多类型数据库，如何在使用 Raw SQL 的时候判断当前使用的数据库类型。

使用 Ormer 的 Driver 方法 可以进行判断。

view设计

模板语法指南

模板处理

模板函数

静态文件处理

模板分页处理

模板语法指南

模板语法指南

beego 模板语法指南

本文讲述 beego 中使用的模板语法，与 go 模板语法基本相同。

基本语法

go 统一使用了 `{{` 和 `}}` 作为左右标签，没有其他的标签符号。如果您想要修改为其它符号，可以参考 模板标签。

使用 `.` 来访问当前位置的上下文

使用 `$` 来引用当前模板根级的上下文

使用 `$var` 来访问创建的变量

[more]

模板中支持的 go 语言符号

```
{{"string"}} // 一般 string
{{`raw string`}} // 原始 string
{{'c'}} // byte
{{print nil}} // nil 也被支持
```

模板中的 pipeline

可以是上下文的变量输出，也可以是函数通过管道传递的返回值

```
{{. | FuncA | FuncB | FuncC}}
```

当 pipeline 的值等于:

- false 或 0
- nil 的指针或 interface

- 长度为 0 的 array, slice, map, string

那么这个 pipeline 被认为是空

if ... else ... end

```
{{if pipeline}}{{end}}
```

if 判断时，pipeline 为空时，相当于判断为 False

```
this.Data["IsLogin"] = true
this.Data["IsHome"] = true
this.Data["IsAbout"] = true
```

支持嵌套的循环

```
{{if .IsHome}}
{{else}}
    {{if .IsAbout}}{{end}}
{{end}}
```

也可以使用 else if 进行

```
{{if .IsHome}}
{{else if .IsAbout}}
{{else}}
{{end}}
```

range ... end

```
{{range pipeline}}{{.}}{{end}}
```

pipeline 支持的类型为 array, slice, map, channel

range 循环内部的 `.` 改变为以上类型的子元素

对应的值长度为 0 时，range 不会执行， `.` 不会改变

```
pages := []struct {
    Num int
}{{10}, {20}, {30}}
```

```
this.Data["Total"] = 100
this.Data["Pages"] = pages
```

使用 `.Num` 输出子元素的 Num 属性，使用 `$.` 引用模板中的根级上下文

```
{{range .Pages}}
    {{.Num}} of {{$.Total}}
{{end}}
```

使用创建的变量，在这里和 go 中的 range 用法是相同的。

```
{{range $index, $elem := .Pages}}
    {{$index}} - {{$elem.Num}} - {{.Num}} of {{$.Total}}
{{end}}
```

range 也支持 else

```
{{range .Pages}}
{{else}}
    {{/* 当 .Pages 为空 或者 长度为 0 时会执行这里 */}}
{{end}}
```

with ... end

```
{{with pipeline}}{{end}}
```

with 用于重定向 pipeline

```
{{with .Field.NestField.SubField}}
    {{.Var}}
{{end}}
```

也可以对变量赋值操作

```
{{with $value := "My name is %s"}}
    {{printf . "slene"}}
{{end}}
```

with 也支持 else

```
{{with pipeline}}
{{else}}
    {{/* 当 pipeline 为空时会执行这里 */}}
{{end}}
```

define

define 可以用来定义自模板，可用于模块定义和模板嵌套

```
{{define "loop"}}
    <li>{{.Name}}</li>
{{end}}
```

使用 template 调用模板

```
<ul>
    {{range .Items}}
        {{template "loop" .}}
    {{end}}
</ul>
```

template

```
{{template "模板名" pipeline}}
```

将对应的上下文 pipeline 传给模板，才可以在模板中调用

Beego 中支持直接载入文件模板

```
{{template "path/to/head.html" .}}
```

Beego 会依据你设置的模板路径读取 head.html

在模板中可以接着载入其他模板，对于模板的分模块处理很有用处

注释

允许多行文本注释，不允许嵌套

```
{{/* comment content
support new line */}}
```

基本函数

变量可以使用符号 `|` 在函数间传递

```
{{.Con | markdown | addlinks}}
```

```
{{.Name | printf "%s"}}
```

使用括号

```
{{printf "nums is %s %d" (printf "%d %d" 1 2) 3}}
```

and

```
{{and .X .Y .Z}}
```

and 会逐一判断每个参数，将返回第一个为空的参数，否则就返回最后一个非空参数

call

```
{{call .Field.Func .Arg1 .Arg2}}
```

call 可以调用函数，并传入参数

调用的函数需要返回 1 个值 或者 2 个值，返回两个值时，第二个值用于返回 error 类型的错误。返回的错误不等于 nil 时，执行将终止。

index

index 支持 map, slice, array, string，读取指定类型对应下标的值

```
this.Data["Maps"] = map[string]string{"name": "Beego"}
```

```
{{index .Maps "name"}}
```

len

```
{{printf "The content length is %d" (.Content|len)}}
```

返回对应类型的长度，支持类型：map, slice, array, string, chan

not

not 返回输入参数的否定值，if true then false else true

or

```
{{or .X .Y .Z}}
```

or 会逐一判断每个参数，将返回第一个非空的参数，否则就返回最后一个参数

print

对应 fmt.Sprint

printf

对应 fmt.Sprintf

println

对应 fmt.Println

urlquery

```
{{urlquery "http://beego.me"}}
```

将返回

```
http%3A%2F%2Fbeego.me
```

eq / ne / lt / le / gt / ge

这类函数一般配合在 if 中使用

`eq` : arg1 == arg2
`ne` : arg1 != arg2
`lt` : arg1 < arg2
`le` : arg1 <= arg2
`gt` : arg1 > arg2
`ge` : arg1 >= arg2

eq 和其他函数不一样的地方是，支持多个参数，和下面的逻辑判断相同

```
arg1==arg2 || arg1==arg3 || arg1==arg4 ...
```

与 if 一起使用

```
{{if eq true .Var1 .Var2 .Var3}}{{end}}
```

```
{{if lt 100 200}}{{end}}
```

> 更多文档请访问 beego 官网。

模板处理

模板处理

beego 的模板处理引擎采用的是 Go 内置的 `html/template` 包进行处理，而且 beego 的模板处理逻辑是采用了缓存编译方式，也就是所有的模板会在 beego 应用启动的时候全部编译然后缓存在 map 里面。

模板目录

beego 中默认的模板目录是 `views` ，用户可以把模板文件放到该目录下，beego 会自动在该目录下的所有模板文件进行解析并缓存，开发模式下每次都会重新解析，不做缓存。当然，用户也可以通过如下的方式改变模板的目录（只能指定一个目录为模板目录）：

```
beego.ViewsPath = "myviewpath"
```

自动渲染

用户无需手动的调用渲染输出模板，beego 会自动的在调用完相应的 method 方法之后调用 Render 函数，当然如果您的应用是不需要模板输出的，那么可以在配置文件或者在 `main.go` 中设置关闭自动渲染。

配置文件配置如下：

```
autorender = false
```

main.go 文件中设置如下：

```
beego.AutoRender = false
```

模板标签

Go 语言的默认模板采用了 `{{` 和 `}}` 作为左右标签，但是我们有时候在开发中可能界面是采用了 AngularJS 开发，他的模板也是这个标签，故而引起了冲突。在 beego 中你可以通过配置文件或者直接设置配置变量修改：

```
beego.TemplateLeft = "<<<"
beego.TemplateRight = ">>>"
```

模板数据

模板中的数据是通过在 Controller 中 `this.Data` 获取的，所以如果你想在模板中获取内容 `{{.Content}}` ,那么你需要在 Controller 中如下设置：

```
this.Data["Content"] = "value"
```

如何使用各种类型的数据渲染：

- 结构体

  结构体结构

  ```
  type A struct{
      Name string
      Age  int
  }
  ```

  控制器数据赋值

  ```
  this.Data["a"]=&A{Name:"astaxie",Age:25}
  ```

  模板渲染数据如下：

  ```
  the username is {{.a.Name}}
  the age is {{.a.Age}}
  ```

- map

  控制器数据赋值

  ```
  mp["name"]="astaxie"
  mp["nickname"] = "haha"
  this.Data["m"]=mp
  ```

  模板渲染数据如下：

```
the username is {{.m.name}}
the username is {{.m.nickname}}
```

- slice

  控制器数据赋值

```
ss :=[]string{"a","b","c"}
this.Data["s"]=ss
```

  模板渲染数据如下：

```
{{range $key, $val := .s}}
{{$key}}
{{$val}}
{{end}}
```

模板名称

beego 采用了 Go 语言内置的模板引擎，所有模板的语法和 Go 的一模一样，至于如何写模板文件，详细的请参考 模板教程。

用户通过在 Controller 的对应方法中设置相应的模板名称，beego 会自动的在 viewpath 目录下查询该文件并渲染，例如下面的设置，beego 会在 admin 下面找 add.tpl 文件进行渲染：

```
this.TplName = "admin/add.tpl"
```

我们看到上面的模板后缀名是 tpl，beego 默认情况下支持 tpl 和 html 后缀名的模板文件，如果你的后缀名不是这两种，请进行如下设置：

```
beego.AddTemplateExt("你文件的后缀名")
```

当你设置了自动渲染，然后在你的 Controller 中没有设置任何的 TplName，那么 beego 会自动设置你的模板文件如下：

```
c.TplName = strings.ToLower(c.controllerName) + "/" + strings.ToLower(c.actionName) + "." + c.TplExt
```

也就是你对应的 Controller 名字+请求方法名.模板后缀，也就是如果你的 Controller 名是 `AddController`，请求方法是 `POST`，默认的文件后缀是 `tpl`，那么就会默认请求 `/viewpath/AddController/post.tpl` 文件。

Layout 设计

beego 支持 layout 设计，例如你在管理系统中，整个管理界面是固定的，只会变化中间的部分，那么你可以通过如下的设置：

```
this.Layout = "admin/layout.html"
this.TplName = "admin/add.tpl"
```

在 layout.html 中你必须设置如下的变量：

```
{{.LayoutContent}}
```

beego 就会首先解析 TplName 指定的文件，获取内容赋值给 LayoutContent，然后最后渲染 layout.html 文件。

目前采用首先把目录下所有的文件进行缓存，所以用户还可以通过类似这样的方式实现 layout：

```
{{template "header.html" .}}
Logic code
{{template "footer.html" .}}
```

> s="default">
>
> s="default">
>
> 特别注意后面的 `.` ,这是传递当前参数到子模板

LayoutSection

对于一个复杂的 `LayoutContent`，其中可能包括有javascript脚本、CSS 引用等，根据惯例，通常 css 会放到 Head 元素中，javascript 脚本需要放到 body 元素的末尾，而其它内容则根据需要放在合适的位置。在 `Layout` 页中仅有一个 `LayoutContent` 是不够的。所以在 `Controller` 中增加了一个 `LayoutSections` 属性，可以允许 `Layout` 页中设置多个 `section`，然后每个 `section` 可以分别包含各自的子模板页。

layout_blog.tpl:

```
<!DOCTYPE html>
<html>
<head>
    <title>Lin Li</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="http://netdna.bootstrapcdn.com/bootstrap/3.0.3/
css/bootstrap.min.css">
    <link rel="stylesheet" href="http://netdna.bootstrapcdn.com/bootstrap/3.0.3/
css/bootstrap-theme.min.css">
    {{.HtmlHead}}
</head>
<body>

    <div class="container">
        {{.LayoutContent}}
    </div>
    <div>
        {{.SideBar}}
    </div>
    <script type="text/javascript" src="http://code.jquery.com/jquery-2.0.3.min.
js"></script>
    <script src="http://netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.mi
n.js"></script>
    {{.Scripts}}
</body>
</html>
```

html_head.tpl:

```
<style>
    h1 {
        color: red;
    }
</style>
```

scripts.tpl：

```
<script type="text/javascript">
    $(document).ready(function() {
        // bla bla bla
    });
</script>
```

逻辑处理如下所示：

```
type BlogsController struct {
    beego.Controller
}

func (this *BlogsController) Get() {
    this.Layout = "layout_blog.tpl"
    this.TplName = "blogs/index.tpl"
    this.LayoutSections = make(map[string]string)
    this.LayoutSections["HtmlHead"] = "blogs/html_head.tpl"
    this.LayoutSections["Scripts"] = "blogs/scripts.tpl"
    this.LayoutSections["Sidebar"] = ""
}
```

renderform 使用

定义 struct:

```
type User struct {
    Id    int          `form:"-"`
    Name  interface{}  `form:"username"`
    Age   int          `form:"age,text,年龄："`
    Sex   string
    Intro string `form:",textarea"`
}
```

- StructTag 的定义用的标签用为 `form`，和 ParseForm 方法 共用一个标签，标签后面有三个可选参数，用 `,` 分割。第一个参数为表单中类型的 `name` 的值，如果为空，则以 `struct field name` 为值。第二个参数为表单组件的类型，如果为空，则为 `text`。表单组件的标签默认为 `struct field name` 的值，否则为第三个值。
- 如果 `form` 标签只有一个值，则为表单中类型 `name` 的值，除了最后一个值可以忽略外，其他位置的必须要有 `,` 号分割，如：`form:",,姓名："`
- 如果要忽略一个字段，有两种办法，一是：字段名小写开头，二是：`form` 标签的值设置为 `-`
- 现在的代码版本只能实现固定的格式，用 br 标签实现换行，无法实现 css 和 class 等代码的插入。所以，要实现 form 的高级排版，不能使用 renderform 的方法，而需要手动处理每一个字段。

controller：

```
func (this *AddController) Get() {
    this.Data["Form"] = &User{}
    this.TplName = "index.tpl"
}
```

Form 的参数必须是一个 struct 的指针。

template:

```
<form action="" method="post">
{{.Form | renderform}}
</form>
```

上面的代码生成的表单为：

```
    Name: <input name="username" type="text" value="test"></br>
    年龄：<input name="age" type="text" value="0"></br>
    Sex: <input name="Sex" type="text" value=""></br>
    Intro: <input name="Intro" type="textarea" value="">
```

模板函数

模板函数

beego 支持用户定义模板函数，但是必须在 `beego.Run()` 调用之前，设置如下：

```
    func hello(in string)(out string){
        out = in + "world"
        return
    }

    beego.AddFuncMap("hi",hello)
```

定义之后你就可以在模板中这样使用了：

```
    {{.Content | hi}}
```

目前 beego 内置的模板函数如下所示：

* dateformat

    实现了时间的格式化，返回字符串，使用方法 {{dateformat .Time "2006-01-02T15:04:05Z07:00"}}。

* date

    实现了类似 PHP 的 date 函数，可以很方便的根据字符串返回时间，使用方法 {{date .T "Y-m-d H:i:s"}}。

* compare

    实现了比较两个对象的比较，如果相同返回 true，否者 false，使用方法 {{compare .A .B}}。

* substr

    实现了字符串的截取，支持中文截取的完美截取，使用方法 {{substr .Str 0 30}}。

* html2str

    实现了把 html 转化为字符串，剔除一些 script、css 之类的元素，返回纯文本信息，使用方法 {{html2str .Htmlinfo}}。

* str2html

    实现了把相应的字符串当作 HTML 来输出，不转义，使用方法 {{str2html .Strhtml}}。

* htmlquote

    实现了基本的 html 字符转义，使用方法 {{htmlquote .quote}}。

* htmlunquote

    实现了基本的反转移字符，使用方法 {{htmlunquote .unquote}}。

* renderform

    根据 StructTag 直接生成对应的表单，使用方法 {{&struct | renderform}}。

* assets_css

    为css文件生成一个`<link>`标签`. `使用方法 {{assets_css src}}

* assets_js

    为js文件生成一个`<script>`标签`.`使用方法 {{assets_js src}}

* config

    获取AppConfig的值`.`使用方法 {{config configType configKey defaultValue}}`.`可选的 configType有String,Bool,Int,Int64,Float,DIY

* map_get

    获取 `map` 的值

    用法:

```
        // In controller
        Data["m"] = map[string]interface{} {
            "a": 1,
            "1": map[string]float64{
                "c": 4,
            },
        }

        // In view
```

```
        {{ map_get .m "a" }} // return 1
        {{ map_get .m 1 "c" }} // return 4

* urlfor

    获取控制器方法的 URL

        {{urlfor "TestController.List"}}

    详见 模板中如何使用
```

静态文件处理

静态文件处理

Go 语言内部其实已经提供了 `http.ServeFile` ，通过这个函数可以实现静态文件的服务。beego 针对这个功能进行了一层封装，通过下面的方式进行静态文件注册：

```
beego.SetStaticPath("/static","public")
```

- 第一个参数是路径，url 路径信息
- 第二个参数是静态文件目录（相对应用所在的目录）

beego 支持多个目录的静态文件注册，用户可以注册如下的静态文件目录：

```
beego.SetStaticPath("/images","images")
beego.SetStaticPath("/css","css")
beego.SetStaticPath("/js","js")
```

设置了如上的静态目录之后，用户访问 `/images/login/login.png` ，那么就会访问应用对应的目录下面的 `images/login/login.png` 文件。如果是访问 `/static/img/logo.png` ，那么就访问 `public/img/logo.png` 文件。

默认情况下 beego 会判断目录下文件是否存在，不存在直接返回 404 页面，如果请求的是 `index.html` ，那么由于 `http.ServeFile` 默认是会跳转的，不提供该页面的显示。因此 beego 可以设置 `beego.BConfig.WebConfig.DirectoryIndex=true` 这样来使得显示 `index.html` 页面。而且开启该功能之后，用户访问目录就会显示该目录下所有的文件列表。

模板分页处理

模板分页处理

分页处理

这里所说的分页，指的是大量数据显示时，每页显示固定的数量的数据，同时显示多个分页链接，用户点击翻页链接或页码时进入到对应的网页。
分页算法中需要处理的问题：
（1）当前数据一共有多少条。
（2）每页多少条，算出总页数。
（3）根据总页数情况，处理翻页链接。
（4）对页面上传入的 Get 或 Post 数据，需要从翻页链接中继续向后传。
（5）在页面显示时，根据每页数量和当前传入的页码，设置查询的 Limit 和 Skip，选择需要的数据。
（6）其他的操作，就是在 View 中显示翻页链接和数据列表的问题了。

模板处理过程中经常需要分页，那么如何进行有效的开发和操作呢？
我们开发组针对这个需求开发了如下的例子，希望对大家有用

- 工具类
  https://github.com/beego/wetalk/blob/master/modules/utils/paginator.go
- 模板
  https://github.com/beego/wetalk/blob/master/views/base/paginator.html
- 使用方法
  https://github.com/beego/wetalk/blob/master/routers/base/base.go#L458

beego的模块设计

Session模块

Grace模块

Cache模块

Logs模块

Httplib模块

Context模块

Toolbox模块

Config模块

I18n模块

Session模块

Session模块

特别注意

这个文档是 session 独立模块，即你单独拿这个模块应用于其他应用中，如果你想在 beego 中使用 session，请查看文档session 控制

session 介绍

session 模块是用来存储客户端用户，session 模块目前只支持 cookie 方式的请求，如果客户端不支持 cookie，那么就无法使用该模块。

session 模块参考了 `database/sql` 的引擎写法，采用了一个接口，多个实现的方式。目前实现了 memory、file、Redis 和 MySQL 四种存储引擎。

通过下面的方式安装 session：

```
go get github.com/astaxie/beego/session
```

session 使用

首先你必须导入包：

```
import (
    "github.com/astaxie/beego/session"
)
```

然后你初始化一个全局的变量用来存储 session 控制器：

```
var globalSessions *session.Manager
```

接着在你的入口函数中初始化数据：

```
func init() {
        sessionConfig := &session.ManagerConfig{
    CookieName:"gosessionid",
    EnableSetCookie: true,
    Gclifetime:3600,
    Maxlifetime: 3600,
    Secure: false,
    CookieLifeTime: 3600,
    ProviderConfig: "./tmp",
    }
    globalSessions, _ = session.NewManager("memory",sessionConfig)
    go globalSessions.GC()
}
```

NewManager 函数的参数的函数如下所示

1. 引擎名字，可以是 memory、file、mysql 或 redis。
2. 一个 JSON 字符串,传入 Manager 的配置信息
    i. cookieName: 客户端存储 cookie 的名字。
    ii. enableSetCookie,omitempty: 是否开启 SetCookie,omitempty 这个设置
    iii. gclifetime: 触发 GC 的时间。
    iv. maxLifetime: 服务器端存储的数据的过期时间
    v. secure: 是否开启 HTTPS，在 cookie 设置的时候有 cookie.Secure 设置。
    vi. sessionIDHashFunc: sessionID 生产的函数，默认是 sha1 算法。
    vii. sessionIDHashKey: hash 算法中的 key。
    viii. cookieLifeTime: 客户端存储的 cookie 的时间，默认值是 0，即浏览器生命周期。
    ix. providerConfig: 配置信息，根据不同的引擎设置不同的配置信息，详细的配置请看下面的引擎设置

最后我们的业务逻辑处理函数中可以这样调用：

```
func login(w http.ResponseWriter, r *http.Request) {
    sess, _ := globalSessions.SessionStart(w, r)
    defer sess.SessionRelease(w)
    username := sess.Get("username")
    if r.Method == "GET" {
        t, _ := template.ParseFiles("login.gtpl")
        t.Execute(w, nil)
    } else {
        sess.Set("username", r.Form["username"])
    }
}
```

globalSessions 有多个函数如下所示：

- SessionStart 根据当前请求返回 session 对象
- SessionDestroy 销毁当前 session 对象
- SessionRegenerateId 重新生成 sessionID
- GetActiveSession 获取当前活跃的 session 用户
- SetHashFunc 设置 sessionID 生成的函数
- SetSecure 设置是否开启 cookie 的 Secure 设置

返回的 session 对象是一个 Interface，包含下面的方法

- Set(key, value interface{}) error
- Get(key interface{}) interface{}
- Delete(key interface{}) error
- SessionID() string
- SessionRelease()
- Flush() error

引擎设置

上面已经展示了 memory 的设置，接下来我们看一下其他三种引擎的设置方式：

- mysql

  其他参数一样，只是第四个参数配置设置如下所示，详细的配置请参考 mysql：

  ```
  username:password@protocol(address)/dbname?param=value
  ```

- redis

  配置文件信息如下所示，表示链接的地址，连接池，访问密码，没有保持为空：

  > 注意：若使用redis等引擎作为session backend，请在使用前导入 < _ “github.com/astaxie/beego/session/redis” >
  >
  > 否则会在运行时发生错误，使用其他引擎时也是同理。

  ```
  127.0.0.1:6379,100,astaxie
  ```

- file

  配置文件如下所示，表示需要保存的目录，默认是两级目录新建文件，例如 sessionID 是 `xsnkjklkjjkh27hjh78908` ，那么目录文件应该是 `./tmp/x/s/xsnkjklkjjkh27hjh78908` ：

```
./tmp
```

如何创建自己的引擎

在开发应用中，你可能需要实现自己的 session 引擎，beego 的这个 session 模块设计的时候就是采用了 interface，所以你可以根据接口实现任意的引擎，例如 memcache 的引擎。

```
type SessionStore interface {
    Set(key, value interface{}) error //set session value
    Get(key interface{}) interface{}  //get session value
    Delete(key interface{}) error     //delete session value
    SessionID() string                //back current sessionID
    SessionRelease()                  // release the resource & save data to pro
vider
    Flush() error                     //delete all data
}

type Provider interface {
    SessionInit(maxlifetime int64, savePath string) error
    SessionRead(sid string) (SessionStore, error)
    SessionExist(sid string) bool
    SessionRegenerate(oldsid, sid string) (SessionStore, error)
    SessionDestroy(sid string) error
    SessionAll() int //get all active session
    SessionGC()
}
```

最后需要注册自己写的引擎：

```
func init() {
    Register("own", ownadaper)
}
```

Grace模块

Grace模块

热升级是什么？

热升级是什么呢？了解 nginx 的同学都知道，nginx 是支持热升级的，可以用老进程服务先前链接的链接，使用新进程服务新的链接，即在不停止服务的情况下完成系统的升级与运行参数修改。那么热升级和热编译是不同的概念，热编译是通过监控文件的变化重新编译，然后重启进程，例如 `bee run` 就是这样的工具

热升级有必要吗？

很多人认为 HTTP 的应用有必要支持热升级吗？那么我可以很负责的说非常有必要，不中断服务始终是我们所追求的目标，虽然很多人说可能服务器会坏掉等等，这个是属于高可用的设计范畴，不要搞混了，这个是可预知的问题，所以我们需要避免这样的升级带来的用户不可用。你还在为以前升级搞到凌晨升级而烦恼嘛？那么现在就赶紧拥抱热升级吧。

grace 模块

grace 模块是 beego 新增的一个独立支持热重启的模块。主要的思路来源于：
http://grisha.org/blog/2014/06/03/graceful-restart-in-golang/

Cache模块

Cache模块

缓存模块

beego 的 cache 模块是用来做数据缓存的，设计思路来自于 `database/sql` ，目前支持 file、memcache、memory 和 redis 四种引擎，安装方式如下：

```
go get github.com/astaxie/beego/cache
```

> s="default">
>
> s="default">
>
> 如果你使用memcache 或者 redis 驱动就需要手工安装引入包

```
go get -u github.com/astaxie/beego/cache/memcache
```

> s="default">
>
> s="default">
>
> 而且需要在使用的地方引入包

```
import _ "github.com/astaxie/beego/cache/memcache"
```

使用入门

首先引入包：

```
import (
    "github.com/astaxie/beego/cache"
)
```

然后初始化一个全局变量对象：

```
bm, err := cache.NewCache("memory", `{"interval":60}`)
```

然后我们就可以使用bm增删改缓存：

```
bm.Put("astaxie", 1, 10*time.Second)
bm.Get("astaxie")
bm.IsExist("astaxie")
bm.Delete("astaxie")
```

引擎设置

目前支持四种不同的引擎，接下来分别介绍这四种引擎如何设置：

- memory

  配置信息如下所示，配置的信息表示 GC 的时间，表示每个 60s 会进行一次过期清理：

  ```
  {"interval":60}
  ```

- file

  配置信息如下所示，配置 `CachePath` 表示缓存的文件目录，`FileSuffix` 表示文件后缀，`DirectoryLevel` 表示目录层级，`EmbedExpiry` 表示过期设置

  ```
  {"CachePath":"./cache","FileSuffix":".cache","DirectoryLevel":"2","EmbedExpiry":"120"}
  ```

- redis

  配置信息如下所示，redis 采用了库 redigo:

  ```
  {"key":"collectionName","conn":":6039","dbNum":"0","password":"thePassWord"}
  ```

  - key: Redis collection 的名称
  - conn: Redis 连接信息
  - dbNum: 连接 Redis 时的 DB 编号. 默认是0.
  - password: 用于连接有密码的 Redis 服务器.

- memcache

  配置信息如下所示，memcache 采用了 vitess的库，表示 memcache 的连接地址：

```
  {"conn":"127.0.0.1:11211"}
```

开发自己的引擎

cache 模块采用了接口的方式实现，因此用户可以很方便的实现接口，然后注册就可以实现自己的 Cache 引擎：

```
type Cache interface {
    Get(key string) interface{}
    GetMulti(keys []string) []interface{}
    Put(key string, val interface{}, timeout time.Duration) error
    Delete(key string) error
    Incr(key string) error
    Decr(key string) error
    IsExist(key string) bool
    ClearAll() error
    StartAndGC(config string) error
}
```

用户开发完毕在最后写类似这样的：

```
func init() {
    Register("myowncache", NewOwnCache())
}
```

Logs模块

Logs模块

日志处理

这是一个用来处理日志的库，它的设计思路来自于 `database/sql` ，目前支持的引擎有file、console、net、smtp，可以通过如下方式进行安装：

```
go get github.com/astaxie/beego/logs
```

如何使用

通用方式

首先引入包：

```
import (
    "github.com/astaxie/beego/logs"
)
```

然后添加输出引擎（log 支持同时输出到多个引擎），这里我们以 console 为例，第一个参数是引擎名（包括：console、file、conn、smtp、es、multifile）

```
logs.SetLogger("console")
```

添加输出引擎也支持第二个参数,用来表示配置信息，详细的配置请看下面介绍：

```
logs.SetLogger(logs.AdapterFile,`{"filename":"project.log","level":7,"maxlines":0,"maxsize":0,"daily":true,"maxdays":10,"color":true}`)
```

然后我们就可以在我们的逻辑中开始任意的使用了：

```
package main

import (
```

```
    "github.com/astaxie/beego/logs"
)

func main() {
    //an official log.Logger
    l := logs.GetLogger()
    l.Println("this is a message of http")
    //an official log.Logger with prefix ORM
    logs.GetLogger("ORM").Println("this is a message of orm")

    logs.Debug("my book is bought in the year of ", 2016)
     logs.Info("this %s cat is %v years old", "yellow", 3)
     logs.Warn("json is a type of kv like", map[string]int{"key": 2016})
       logs.Error(1024, "is a very", "good game")
       logs.Critical("oh,crash")
}
```

多个实例

一般推荐使用通用方式进行日志，但依然支持单独声明来使用独立的日志

```
    package main

    import (
        "github.com/astaxie/beego/logs"
    )

    func main() {
        log := logs.NewLogger()
        log.SetLogger(logs.AdapterConsole)
        log.Debug("this is a debug message")
    }
```

输出文件名和行号

日志默认不输出调用的文件名和文件行号,如果你期望输出调用的文件名和文件行号,可以如下设置

```
logs.EnableFuncCallDepth(true)
```

开启传入参数 true,关闭传入参数 false,默认是关闭的.

如果你的应用自己封装了调用 log 包,那么需要设置 SetLogFuncCallDepth,默认是 2,也就是直接调用的层级,如果你封装了多层,那么需要根据自己的需求进行调整.

```
logs.SetLogFuncCallDepth(3)
```

异步输出日志

为了提升性能, 可以设置异步输出:

```
logs.Async()
```

异步输出允许设置缓冲 chan 的大小

```
logs.Async(1e3)
```

引擎配置设置

- console

  命令行输出，默认输出到`os.Stdout`：

  ```
  logs.SetLogger(logs.AdapterConsole, `{"level":1,"color":true}`)
  ```

  主要的参数如下说明：

  - level 输出的日志级别
  - color 是否开启打印日志彩色打印(需环境支持彩色输出)

- file

  设置的例子如下所示：

  ```
  logs.SetLogger(logs.AdapterFile, `{"filename":"test.log"}`)
  ```

  主要的参数如下说明：

  - filename 保存的文件名
  - maxlines 每个文件保存的最大行数，默认值 1000000

- maxsize 每个文件保存的最大尺寸，默认值是 1 << 28, //256 MB
- daily 是否按照每天 logrotate，默认是 true
- maxdays 文件最多保存多少天，默认保存 7 天
- rotate 是否开启 logrotate，默认是 true
- level 日志保存的时候的级别，默认是 Trace 级别
- perm 日志文件权限

- multifile

  设置的例子如下所示：

```
  logs.SetLogger(logs.AdapterMultiFile, `{"filename":"test.log","separate":[
"emergency", "alert", "critical", "error", "warning", "notice", "info", "deb
ug"]}`)
```

  主要的参数如下说明(除 separate 外,均与file相同)：

  - filename 保存的文件名
  - maxlines 每个文件保存的最大行数，默认值 1000000
  - maxsize 每个文件保存的最大尺寸，默认值是 1 << 28, //256 MB
  - daily 是否按照每天 logrotate，默认是 true
  - maxdays 文件最多保存多少天，默认保存 7 天
  - rotate 是否开启 logrotate，默认是 true
  - level 日志保存的时候的级别，默认是 Trace 级别
  - perm 日志文件权限
  - separate 需要单独写入文件的日志级别,设置后命名类似 test.error.log

- conn

  网络输出，设置的例子如下所示：

```
  logs.SetLogger(logs.AdapterConn, `{"net":"tcp","addr":":7020"}`)
```

  主要的参数说明如下：

  - reconnectOnMsg 是否每次链接都重新打开链接，默认是 false
  - reconnect 是否自动重新链接地址，默认是 false

  - net 发开网络链接的方式，可以使用 tcp、unix、udp 等
  - addr 网络链接的地址
  - level 日志保存的时候的级别，默认是 Trace 级别

- smtp

  邮件发送，设置的例子如下所示：

```
  logs.SetLogger(logs.AdapterMail, `{"username":"beegotest@gmail.com","passw
ord":"xxxxxxxx","host":"smtp.gmail.com:587","sendTos":["xiemengjun@gmail.com
"]}`)
```

  主要的参数说明如下：

  - username smtp 验证的用户名
  - password smtp 验证密码
  - host 发送的邮箱地址
  - sendTos 邮件需要发送的人，支持多个
  - subject 发送邮件的标题，默认是 `Diagnostic message from server`
  - level 日志发送的级别，默认是 Trace 级别

- ElasticSearch

  输出到 ElasticSearch:

```
    logs.SetLogger(logs.AdapterEs, `{"dsn":"http://localhost:9200/","level"
:1}`)
```

- 简聊

  输出到简聊：

```
  logs.SetLogger(logs.AdapterJianLiao, `{"authorname":"xxx","title":"beego",
"webhookurl":"https://jianliao.com/xxx", "redirecturl":"https://jianliao.co
m/xxx","imageurl":"https://jianliao.com/xxx","level":1}`)
```

- slack

输出到slack:

```
 logs.SetLogger(logs.AdapterSlack, `{"webhookurl":"https://slack.com/xxx","level":1}`)
```

Httplib模块

Httplib模块

客户端请求

httplib 库主要用来模拟客户端发送 HTTP 请求，类似于 Curl 工具，支持 JQuery 类似的链式操作。使用起来相当的方便；通过如下方式进行安装：

```
go get github.com/astaxie/beego/httplib
```

如何使用

首先导入包

```
import (
    "github.com/astaxie/beego/httplib"
)
```

然后初始化请求方法，返回对象

```
req := httplib.Get("http://beego.me/")
```

然后我们就可以获取数据了

```
str, err := req.String()
if err != nil {
    t.Fatal(err)
}
fmt.Println(str)
```

支持的方法对象

httplib 包里面支持如下的方法返回 request 对象：

- Get(url string)

- Post(url string)
- Put(url string)
- Delete(url string)
- Head(url string)

支持 debug 输出

可以根据上面五个方法返回的对象进行调试信息的输出：

```
req.Debug(true)
```

这样就可以看到请求数据的详细输出

```
httplib.Get("http://beego.me/").Debug(true).Response()

//输出数据如下
GET / HTTP/0.0
Host: beego.me
User-Agent: beegoServer
```

支持 HTTPS 请求

如果请求的网站是 HTTPS 的，那么我们就需要设置 client 的 TLS 信息，如下所示：

```
req.SetTLSClientConfig(&tls.Config{InsecureSkipVerify: true})
```

关于如何设置这些信息请访问： http://gowalker.org/crypto/tls#Config

支持超时设置

通过如下接口可以设置请求的超时时间和数据读取时间：

```
req.SetTimeout(connectTimeout, readWriteTimeout)
```

以上方法都是针对 request 对象的，所以你第一步必须是返回 request 对象，然后链式操作，类似这样的代码：

```
httplib.Get("http://beego.me/").SetTimeout(100 * time.Second, 30 * time.Second).
Response()
```

设置请求参数

对于 Put 或者 Post 请求，需要发送参数，那么可以通过 Param 发送 k/v 数据，如下所示：

```
req := httplib.Post("http://beego.me/")
req.Param("username","astaxie")
req.Param("password","123456")
```

发送大片的数据

有时候需要上传文件之类的模拟，那么如何发送这个文件数据呢？可以通过 Body 函数来操作，举例如下：

```
req := httplib.Post("http://beego.me/")
bt,err:=ioutil.ReadFile("hello.txt")
if err!=nil{
    log.Fatal("read file err:",err)
}
req.Body(bt)
```

设置 header 信息

除了请求参数之外，我们有些时候需要模拟一些头信息，例如

```
Accept-Encoding:gzip,deflate,sdch
Host:beego.me
User-Agent:Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_0) AppleWebKit/537.36 (KH
TML, like Gecko) Chrome/31.0.1650.57 Safari/537.36
```

可以通过 Header 函数来设置，如下所示：

```
req := httplib.Post("http://beego.me/")
req.Header("Accept-Encoding","gzip,deflate,sdch")
req.Header("Host","beego.me")
req.Header("User-Agent","Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_0) AppleWeb
Kit/537.36 (KHTML, like Gecko) Chrome/31.0.1650.57 Safari/537.36")
```

设置 transport

http请求的传输由 `http.RoundTrip` 承载，因此我们可以实现接口以实现链接的控制。通过设置，我们可以实现长连接，如下所示:

```
var tp http.RoundTripper = &http.Transport{
    DialContext: (&net.Dialer{
        Timeout:   30 * time.Second,
        KeepAlive: 30 * time.Second,
        DualStack: true,
    }).DialContext,
    MaxIdleConns:          100,
    IdleConnTimeout:       90 * time.Second,
    ExpectContinueTimeout: 1 * time.Second,
}

req := httplib.Post("http://beego.me/")
req.SetTransport(tp)
```

httplib支持文件直接上传接口

PostFile 第一个参数是 form 表单的字段名,第二个是需要发送的文件名或者文件路径

```
b:=httplib.Post("http://beego.me/")
b.Param("username","astaxie")
b.Param("password","123456")
b.PostFile("uploadfile1", "httplib.pdf")
b.PostFile("uploadfile2", "httplib.txt")
str, err := b.String()
if err != nil {
    t.Fatal(err)
}
```

获取返回结果

上面这些都是在发送请求之前的设置，接下来我们开始发送请求，然后如何来获取数据呢？主要有如下几种方式：

- 返回 Response 对象， `req.Response()` 方法

  这个是 http.Response 对象，用户可以自己读取 body 的数据等。

- 返回 bytes, `req.Bytes()` 方法

直接返回请求 URL 返回的内容

- 返回 string， `req.String()` 方法

  直接返回请求 URL 返回的内容

- 保存为文件， `req.ToFile(filename)` 方法

  返回结果保存到文件名为 filename 的文件中

- 解析为 JSON 结构， `req.ToJSON(&result)` 方法

  返回结构直接解析为 JSON 格式，解析到 result 对象中

- 解析为 XML 结构， `req.ToXml(&result)` 方法

  返回结构直接解析为 XML 格式，解析到 result 对象中

Context模块

Context模块

上下文模块

上下文模块主要是针对 HTTP 请求中，request 和 response 的进一步封装，他包括用户的输入和输出，用户的输入即为 request，context 模块中提供了 Input 对象进行解析，用户的输出即为 response，context 模块中提供了 Output 对象进行输出。

context 对象

context 对象是对 Input 和 Output 的封装，里面封装了几个方法：

- Redirect
- Abort
- WriteString
- GetCookie
- SetCookie

context 对象是 Filter 函数的参数对象，这样你就可以通过 filter 来修改相应的数据，或者提前结束整个的执行过程。

Input 对象

Input 对象是针对 request 的封装，里面通过 reqeust 实现很多方便的方法，具体如下：

- Protocol

  获取用户请求的协议，例如 `HTTP/1.0`

- Uri

  用户请求的 RequestURI，例如 `/hi?id=1001`

- Url

请求的 URL 地址，例如 `/hi`

- Site

  请求的站点地址,scheme+doamin 的组合，例如 `http://beego.me`

- Scheme

  请求的 scheme，例如 “http” 或者 “https”

- Domain

  请求的域名，例如 `beego.me`

- Host

  请求的域名，和 domain 一样

- Method

  请求的方法，标准的 HTTP 请求方法，例如 `GET` 、 `POST` 等

- Is

  判断是否是某一个方法，例如 `Is("GET")` 返回 true

- IsAjax

  判断是否是 AJAX 请求，如果是返回 true，不是返回 false

- IsSecure

  判断当前请求是否 HTTPS 请求，是返回 true，否返回 false

- IsWebsocket

  判断当前请求是否 Websocket 请求，如果是返回 true，否返回 false

- IsUpload

  判断当前请求是否有文件上传，有返回 true，否返回 false

- IP

  返回请求用户的 IP，如果用户通过代理，一层一层剥离获取真实的 IP

- Proxy

  返回用户代理请求的所有 IP

- Refer

  返回请求的 refer 信息

- SubDomains

  返回请求域名的根域名，例如请求是 `blog.beego.me` ，那么调用该函数返回 `beego.me`

- Port

  返回请求的端口，例如返回 8080

- UserAgent

  返回请求的 `UserAgent` ，例如 `Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/31.0.1650.57 Safari/537.36`

- Param

  在路由设置的时候可以设置参数，这个是用来获取那些参数的，例如 `Param(":id")` ，返回12

- Query

  该函数返回 Get 请求和 Post 请求中的所有数据，和 PHP 中 `$_REQUEST` 类似

- Header

  返回相应的 header 信息，例如 `Header("Accept-Language")` ，就返回请求头中对应的信息 `zh-CN,zh;q=0.8,en;q=0.6`

- Cookie

  返回请求中的 cookie 数据，例如 `Cookie("username")` ，就可以获取请求头中携带的 cookie 信息中 username 对应的值

- Session

  session 是用户可以初始化的信息，默认采用了 beego 的 session 模块中的 Session 对象，用来获取存储在服务器端中的数据。

- Body

  返回请求 Body 中数据，例如 API 应用中，很多用户直接发送 json 数据包，那么通过 Query 这种函数无法获取数据，就必须通过该函数获取数据。该函数已经 beego 1.5 版本之后删除，目前可以通过 RequestBody 获取数据。

- GetData

  用来获取 Input 中 Data 中的数据

- SetData

  用来设置 Input 中 Data 的值，上面 GetData 和这个函数都是用来方便用户在 Filter 中传递数据到 Controller 中来执行

Output 对象

Output 是针对 Response 的封装，里面提供了很多方便的方法：

- Header

  设置输出的 header 信息，例如 `Header("Server","beego")`

- Body

  设置输出的内容信息，例如 `Body([]byte("astaxie"))`

- Cookie

  设置输出的 cookie 信息，例如 `Cookie("sessionID","beegoSessionID")`

- Json

  把 Data 格式化为 Json，然后调用 Body 输出数据

- Jsonp

  把 Data 格式化为 Jsonp，然后调用 Body 输出数据

- Xml

  把 Data 格式化为 Xml，然后调用 Body 输出数据

- Download

  把 file 路径传递进来，然后输出文件给用户

- ContentType

  设置输出的 ContentType

- SetStatus

  设置输出的 status

- Session

  设置在服务器端保存的值，例如 `Session("username","astaxie")` ，这样用户就可以在下次使用的时候读取

- IsCachable

  根据 status 判断，是否为缓存类的状态

- IsEmpty

  根据 status 判断，是否为输出内容为空的状态

- IsOk

  根据 status 判断，是否为 200 的状态

- IsSuccessful

  根据 status 判断，是否为正常的状态

- IsRedirect

  根据 status 判断，是否为跳转类的状态

- IsForbidden

  根据 status 判断，是否为禁用类的状态

- IsNotFound

  根据 status 判断，是否为找不到资源类的状态

- IsClientError

  根据 status 判断，是否为请求客户端错误的状态

- IsServerError

  根据 status 判断，是否为服务器端错误的状态

Toolbox模块

Toolbox模块

核心工具模块

这个模块主要是参考了 Dropwizard 框架，是一位用户提醒我说有这么一个框架，然后里面实现一些很酷的东西。那个 issue 详细描述了该功能的雏形，然后就在参考该功能的情况下增加了一些额外的很酷的功能，接下来我将一一介绍这个模块中的几个功能：健康检查、性能调试、访问统计、计划任务。

如何安装

```
go get github.com/astaxie/beego/toolbox
```

healthcheck

监控检查是用于当你应用于产品环境中进程，检查当前的状态是否正常，例如你要检查当前数据库是否可用，如下例子所示：

```
type DatabaseCheck struct {
}

func (dc *DatabaseCheck) Check() error {
    if dc.isConnected() {
        return nil
    } else {
        return errors.New("can't connect database")
    }
}
```

然后就可以通过如下方式增加检测项：

```
toolbox.AddHealthCheck("database",&DatabaseCheck{})
```

加入之后，你可以往你的管理端口 `/healthcheck` 发送GET请求：

```
$ curl http://beego.me:8088/healthcheck
* deadlocks: OK
* database: OK
```

如果检测显示是正确的，那么输出 OK，如果检测出错，显示出错的信息。

profile

对于运行中的进程的性能监控是我们进行程序调优和查找问题的最佳方法，例如 GC、goroutine 等基础信息。profile 提供了方便的入口方便用户来调试程序，他主要是通过入口函数 `ProcessInput` 来进行处理各类请求，主要包括以下几种调试：

- lookup goroutine

  打印出来当前全部的 goroutine 执行的情况，非常方便查找各个 goroutine 在做的事情：

```
  goroutine 3 [running]:
  runtime/pprof.writeGoroutineStacks(0x634238, 0xc210000008, 0x62b000, 0xd20
0000000000000)
      /Users/astaxie/go/src/pkg/runtime/pprof/pprof.go:511 +0x7c
  runtime/pprof.writeGoroutine(0x634238, 0xc210000008, 0x2, 0xd2676410957b30
fd, 0xae98)
      /Users/astaxie/go/src/pkg/runtime/pprof/pprof.go:500 +0x3c
  runtime/pprof.(*Profile).WriteTo(0x52ebe0, 0x634238, 0xc210000008, 0x2, 0x
1, ...)
      /Users/astaxie/go/src/pkg/runtime/pprof/pprof.go:229 +0xb4
  _/Users/astaxie/github/beego/toolbox.ProcessInput(0x2c89f0, 0x10, 0x634238
, 0xc210000008)
      /Users/astaxie/github/beego/toolbox/profile.go:26 +0x256
  _/Users/astaxie/github/beego/toolbox.TestProcessInput(0xc21004e090)
      /Users/astaxie/github/beego/toolbox/profile_test.go:9 +0x5a
  testing.tRunner(0xc21004e090, 0x532320)
      /Users/astaxie/go/src/pkg/testing/testing.go:391 +0x8b
  created by testing.RunTests
      /Users/astaxie/go/src/pkg/testing/testing.go:471 +0x8b2

  goroutine 1 [chan receive]:
  testing.RunTests(0x315668, 0x532320, 0x4, 0x4, 0x1)
      /Users/astaxie/go/src/pkg/testing/testing.go:472 +0x8d5
  testing.Main(0x315668, 0x532320, 0x4, 0x4, 0x537700, ...)
      /Users/astaxie/go/src/pkg/testing/testing.go:403 +0x84
  main.main()
      _/Users/astaxie/github/beego/toolbox/_test/_testmain.go:53 +0x9c
```

- lookup heap

  用来打印当前 heap 的信息：

```
heap profile: 1: 288 [2: 296] @ heap/1048576
1: 288 [2: 296] @
```

```
    # runtime.MemStats
    # Alloc = 275504
    # TotalAlloc = 275512
    # Sys = 4069608
    # Lookups = 5
    # Mallocs = 469
    # Frees = 1
    # HeapAlloc = 275504
    # HeapSys = 1048576
    # HeapIdle = 647168
    # HeapInuse = 401408
    # HeapReleased = 0
    # HeapObjects = 468
    # Stack = 24576 / 131072
    # MSpan = 4472 / 16384
    # MCache = 1504 / 16384
    # BuckHashSys = 1476472
    # NextGC = 342976
    # PauseNs = [370712 77378 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
    # NumGC = 2
    # EnableGC = true
    # DebugGC = false
```

- lookup threadcreate

  查看创建线程的信息：

```
threadcreate profile: total 4
1 @ 0x17f68 0x183c7 0x186a8 0x188cc 0x19ca9 0xcf41 0x139a3 0x196c0
#    0x183c7     newm+0x27            /Users/astaxie/go/src/pkg/runtime/pro
```

```
c.c:896
  #    0x186a8    startm+0xb8            /Users/astaxie/go/src/pkg/runtime/proc.c:974
  #    0x188cc    handoffp+0x1ac         /Users/astaxie/go/src/pkg/runtime/proc.c:992
  #    0x19ca9    runtime.entersyscallblock+0x129    /Users/astaxie/go/src/pkg/runtime/proc.c:1514
  #    0xcf41     runtime.notetsleepg+0x71    /Users/astaxie/go/src/pkg/runtime/lock_sema.c:253
  #    0x139a3    runtime.MHeap_Scavenger+0xa3    /Users/astaxie/go/src/pkg/runtime/mheap.c:463

  1 @ 0x17f68 0x183c7 0x186a8 0x188cc 0x189c3 0x1969b 0x2618b
  #    0x183c7    newm+0x27        /Users/astaxie/go/src/pkg/runtime/proc.c:896
  #    0x186a8    startm+0xb8      /Users/astaxie/go/src/pkg/runtime/proc.c:974
  #    0x188cc    handoffp+0x1ac   /Users/astaxie/go/src/pkg/runtime/proc.c:992
  #    0x189c3    stoplockedm+0x83    /Users/astaxie/go/src/pkg/runtime/proc.c:1049
  #    0x1969b    runtime.gosched0+0x8b    /Users/astaxie/go/src/pkg/runtime/proc.c:1382
  #    0x2618b    runtime.mcall+0x4b    /Users/astaxie/go/src/pkg/runtime/asm_amd64.s:178

  1 @ 0x17f68 0x183c7 0x170bc 0x196c0
  #    0x183c7    newm+0x27        /Users/astaxie/go/src/pkg/runtime/proc.c:896
  #    0x170bc    runtime.main+0x3c    /Users/astaxie/go/src/pkg/runtime/proc.c:191

  1 @
```

- lookup block

  查看 block 信息：

```
  --- contention:
  cycles/second=2294781025
```

- start cpuprof

开始记录 cpuprof 信息，生产一个文件 cpu-pid.pprof，开始记录当前进程的 CPU 处理信息

- stop cpuprof

  关闭记录信息

- get memprof

  开启记录 memprof，生产一个文件 mem-pid.memprof

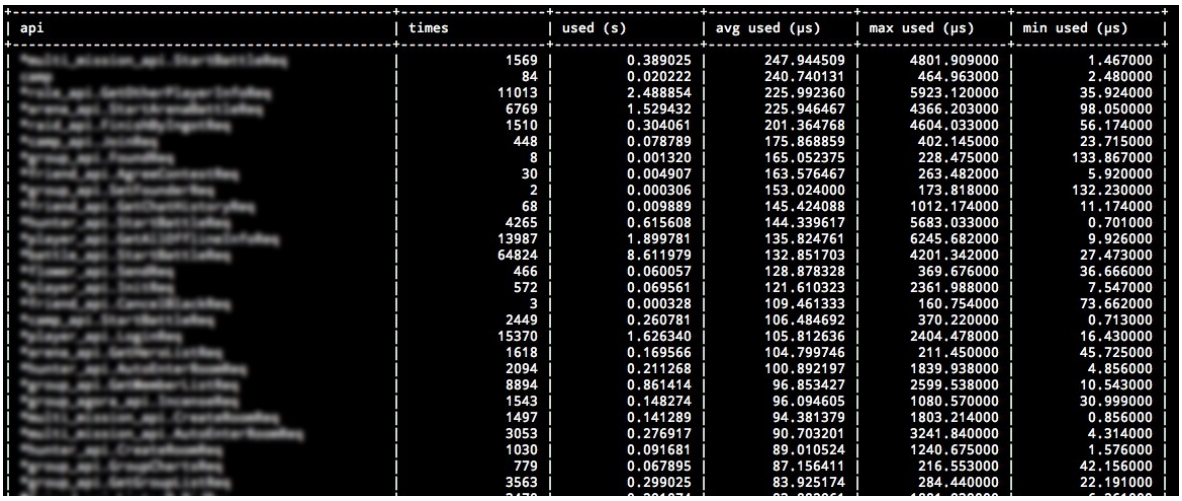
- gc summary

  查看 GC 信息

```
 NumGC:2 Pause:54.54us Pause(Avg):170.82us Overhead:177.49% Alloc:248.97K Sys:3.88M Alloc(Rate):1.23G/s Histogram:287.09us 287.09us 287.09us
```

statistics

请先看下面这张效果图，你有什么想法，很酷？是的，很酷，现在 tootlbox 就是支持这样的功能了：



api	times	used (s)	avg used (µs)	max used (µs)	min used (µs)
[illegible]	1569	0.389025	247.944509	4801.909000	1.467000
[illegible]	84	0.020222	240.740131	464.963000	2.480000
[illegible]	11013	2.488854	225.992360	5923.120000	35.924000
[illegible]	6769	1.529432	225.946467	4366.203000	98.050000
[illegible]	1510	0.304061	201.364768	4604.033000	56.174000
[illegible]	448	0.078789	175.868859	402.145000	23.715000
[illegible]	8	0.001320	165.052375	228.475000	133.867000
[illegible]	30	0.004907	163.576467	263.482000	5.920000
[illegible]	2	0.000306	153.024000	173.818000	132.230000
[illegible]	68	0.009889	145.424088	1012.174000	11.174000
[illegible]	4265	0.615608	144.339617	5683.033000	0.701000
[illegible]	13987	1.899781	135.824761	6245.682000	9.926000
[illegible]	64824	8.611979	132.851703	4201.342000	27.473000
[illegible]	466	0.060057	128.878328	369.676000	36.666000
[illegible]	572	0.069561	121.610323	2361.988000	7.547000
[illegible]	3	0.000328	109.461333	160.754000	73.662000
[illegible]	2449	0.260781	106.484692	370.220000	0.713000
[illegible]	15370	1.626340	105.812636	2404.478000	16.430000
[illegible]	1618	0.169566	104.799746	211.450000	45.725000
[illegible]	2094	0.211268	100.892197	1839.938000	4.856000
[illegible]	8894	0.861414	96.853427	2599.538000	10.543000
[illegible]	1543	0.148274	96.094605	1080.570000	30.999000
[illegible]	1497	0.141289	94.381379	1803.214000	0.856000
[illegible]	3053	0.276917	90.703201	3241.840000	4.314000
[illegible]	1030	0.091681	89.010524	1240.675000	1.576000
[illegible]	779	0.067895	87.156411	216.553000	42.156000
[illegible]	3563	0.299025	83.925174	284.440000	22.191000

如何使用这个统计呢？如下所示添加统计：

```
toolbox.StatisticsMap.AddStatistics("POST", "/api/user", "&admin.user", time.Duration(2000))
toolbox.StatisticsMap.AddStatistics("POST", "/api/user", "&admin.user", time.Duration(120000))
```

```
toolbox.StatisticsMap.AddStatistics("GET", "/api/user", "&admin.user", time.Duration(13000))
toolbox.StatisticsMap.AddStatistics("POST", "/api/admin", "&admin.user", time.Duration(14000))
toolbox.StatisticsMap.AddStatistics("POST", "/api/user/astaxie", "&admin.user", time.Duration(12000))
toolbox.StatisticsMap.AddStatistics("POST", "/api/user/xiemengjun", "&admin.user", time.Duration(13000))
toolbox.StatisticsMap.AddStatistics("DELETE", "/api/user", "&admin.user", time.Duration(1400))
```

获取统计信息

```
toolbox.StatisticsMap.GetMap(os.Stdout)
```

输出如下格式的信息：

```
| requestUrl                                  | method      | times
| used              | max used          | min used          | avg used          |
| /api/user                                   | POST        | 2
| 122.00us          | 120.00us          | 2.00us            | 61.00us           |
| /api/user                                   | GET         | 1
| 13.00us           | 13.00us           | 13.00us           | 13.00us           |
| /api/user                                   | DELETE      | 1
| 1.40us            | 1.40us            | 1.40us            | 1.40us            |
| /api/admin                                  | POST        | 1
| 14.00us           | 14.00us           | 14.00us           | 14.00us           |
| /api/user/astaxie                           | POST        | 1
| 12.00us           | 12.00us           | 12.00us           | 12.00us           |
| /api/user/xiemengjun                        | POST        | 1
| 13.00us           | 13.00us           | 13.00us           | 13.00us           |
```

task

玩过 linux 的用户都知道有一个计划任务的工具 crontab，我们经常利用该工具来定时的做一些任务，但是有些时候我们的进程内也希望定时的来处理一些事情，例如定时的汇报当前进程的内存信息，goroutine 信息等。或者定时的进行手工触发 GC，或者定时的清理一些日志数据等，所以实现了秒级别的定时任务，首先让我们看看如何使用：

1. 初始化一个任务

```
tk1 := toolbox.NewTask("tk1", "0 12 * * * *", func() error { fmt.Println("tk1"); return nil })
```

函数原型：

NewTask(tname string, spec string, f TaskFunc) *Task

- tname 任务名称
- spec 定时任务格式，请参考下面的详细介绍
- f 执行的函数 func() error

2. 可以测试开启运行

   可以通过如下的代码运行 TaskFunc，和 spec 无关，用于检测写的函数是否如预期所希望的这样：

```
err := tk.Run()
if err != nil {
    t.Fatal(err)
}
```

3. 加入全局的计划任务列表

```
toolbox.AddTask("tk1", tk1)
```

4. 开始执行全局的任务

```
toolbox.StartTask()
defer toolbox.StopTask()
```

spec 详解

spec 格式是参照 crontab 做的，详细的解释如下所示：

```
//前6个字段分别表示：
//       秒钟：0-59
//       分钟：0-59
//       小时：1-23
//       日期：1-31
//       月份：1-12
//       星期：0-6（0 表示周日）

//还可以用一些特殊符号：
//       *：  表示任何时刻
//       ,：   表示分割，如第三段里：2,4，表示 2 点和 4 点执行
```

```
//          一：表示一个段，如第三端里： 1-5，就表示 1 到 5 点
//        /n : 表示每个n的单位执行一次，如第三段里，*/1, 就表示每隔 1 个小时执行一次命令。也可以写成1-23/1.
////////////////////////////////////////////////////////
//     0/30 * * * * *                        每 30 秒 执行
//     0 43 21 * * *                         21:43 执行
//     0 15 05 * * *                          05:15 执行
//     0 0 17 * * *                          17:00 执行
//     0 0 17 * * 1                          每周一的 17:00 执行
//     0 0,10 17 * * 0,2,3                   每周日,周二,周三的 17:00和 17:10 执行
//     0 0-10 17 1 * *                       每月1日从 17:00 到 7:10 每隔 1 分钟 执行
//     0 0 0 1,15 * 1                        每月1日和 15 日和 一日的 0:00 执行
//     0 42 4 1 * *                           每月1日的 4:42 分 执行
//     0 0 21 * * 1-6                         周一到周六 21:00 执行
//     0 0,10,20,30,40,50 * * * *             每隔 10 分 执行
//     0 */10 * * * *                           每隔 10 分 执行
//     0 * 1 * * *                               从 1:0 到 1:59 每隔 1 分钟 执行
//     0 0 1 * * *                               1:00 执行
//     0 0 */1 * * *                             每时 0 分 每隔 1 小时 执行
//     0 0 * * * *                               每时 0 分 每隔 1 小时 执行
//     0 2 8-20/3 * * *                        8:02,11:02,14:02,17:02,20:02 执行
//     0 30 5 1,15 * *                         1 日 和 15 日的 5:30 执行
```

调试模块(已移动到utils模块)

我们经常需要打印一些参数进行调试，但是默认的参数打印总是不是很完美，也没办法定位代码行之类的，所以 beego 的 toolbox 模块进行了 debug 模块的开发，主要包括了两个函数：

- Display() 直接打印结果到 console
- GetDisplayString() 返回打印的字符串

两个函数的功能一模一样，第一个是直接打印到 console，第二个是返回字符串，方便用户存储到日志或者其他存储。

使用很方便，是 key/value 串出现的，如下示例：

```
Display("v1", 1, "v2", 2, "v3", 3)
```

打印结果如下：

```
2013/12/16 23:48:41 [Debug] at TestPrint() [/Users/astaxie/github/beego/toolbox/debug_test.go:13]

[Variables]
v1 = 1
v2 = 2
v3 = 3
```

指针类型的打印如下：

```
type mytype struct {
    next *mytype
    prev *mytype
}

var v1 = new(mytype)
var v2 = new(mytype)

v1.prev = nil
v1.next = v2

v2.prev = v1
v2.next = nil

Display("v1", v1, "v2", v2)
```

打印结果如下：

```
2013/12/16 23:48:41 [Debug] at TestPrintPoint() [/Users/astaxie/github/beego/toolbox/debug_test.go:26]

[Variables]
v1 = &toolbox.mytype{
    next: &toolbox.mytype{
        next: nil,
        prev: 0x210335420,
    },
    prev: nil,
}
v2 = &toolbox.mytype{
    next: nil,
    prev: &toolbox.mytype{
        next: 0x210335430,
        prev: nil,
```

```
    },
}
```

Config模块

Config模块

配置文件解析

这是一个用来解析文件的库，它的设计思路来自于 `database/sql` ，目前支持解析的文件格式有 ini、json、xml、yaml，可以通过如下方式进行安装：

```
go get github.com/astaxie/beego/config
```

> s="default">
>
> s="default">
>
> 如果你使用xml 或者 yaml 驱动就需要手工安装引入包

```
go get -u github.com/astaxie/beego/config/xml
```

> s="default">
>
> s="default">
>
> 而且需要在使用的地方引入包

```
import _ "github.com/astaxie/beego/config/xml"
```

如何使用

首先初始化一个解析器对象

```
iniconf, err := NewConfig("ini", "testini.conf")
if err != nil {
    t.Fatal(err)
}
```

然后通过对象获取数据

```
iniconf.String("appname")
```

解析器对象支持的函数有如下：

- Set(key, val string) error
- String(key string) string
- Int(key string) (int, error)
- Int64(key string) (int64, error)
- Bool(key string) (bool, error)
- Float(key string) (float64, error)
- DIY(key string) (interface{}, error)

s="default">

s="default">

ini 配置文件支持 section 操作，key通过 `section::key` 的方式获取

s="default">

s="default">

例如下面这样的配置文件

```
[demo]
key1 = "asta"
key2 = "xie"
```

s="default">

s="default">

那么可以通过 `iniconf.String("demo::key2")` 获取值.

如何获取环境变量

> config 模块支持环境变量配置，对应配置项 Key 格式为 `${环境变量名}` ，则 Value = os.Getenv('环境变量名')。
> 同时可配置默认值，当环境变量无此配置或者环境变量值为空时，则优先使用默认值。包含默认值的 Key 格式为 `${GOPATH||/home/workspace/go/}` ，使用 `||` 分割环境变量和默认值。
>
> **注意** 获取环境变量值仅仅是在配置文件解析时处理，而不会在调用函数获取配置项时实时处理。

I18n模块

I18n模块

国际化介绍

i18n 模块主要用于实现站点或应用的国际化功能，实现多语言界面与反馈，增强用户体验。像 Go Walker 和 beego 官网 即是采用了该模块实现了中文与英文的双语界面。

您可以通过以下方式安装该模块：

```
go get github.com/beego/i18n
```

i18n 使用

首先，您需要导入该包：

```
import (
    "github.com/beego/i18n"
)
```

该模块主要采用的是键值对的形式，非常类似 INI 格式的配置文件，但又在此基础上增强了一些功能。每个语种均对应一个本地化文件，例如 beego 官网的 `conf` 目录下就有 `locale_en-US.ini` 和 `locale_zh-CN.ini` 两个本地化文件。

本地化文件的文件名和后缀是随意的，不过我们建议您采用与 beego 官网相同的风格来对它们命名。

最简实例

下面是两个最简单的本地化文件示例：

文件 `locale_en-US.ini` ：

```
hi = hello
bye = goodbye
```

文件 `locale_zh-CN.ini` ：

```
hi = 您好
bye = 再见
```

在控制器中使用

对于每个请求，beego 都会采用单独的 goroutine 来处理，因此可以对每个控制器匿名嵌入一个 `i18n.Locale` 结构用于处理当前请求的本地化响应。这个要求您能够理解 beego 的 `baseController` 理念和使用 `Prepare` 方法，具体可参考 beego 官网的控制器源码部分 `routers/router.go` 。

接受请求之后，在 `baseController` 的 `Prepare` 方法内进行语言处理，这样便可应用后所有其它控制器内而无需重复编写代码。

注册本地化文件

以下代码摘取自 beego 官网源码 `routers/init.go` ：

```
// Initialized language type list.
langs := strings.Split(beego.AppConfig.String("lang::types"), "|")
names := strings.Split(beego.AppConfig.String("lang::names"), "|")
langTypes = make([]*langType, 0, len(langs))
for i, v := range langs {
    langTypes = append(langTypes, &langType{
        Lang: v,
        Name: names[i],
    })
}

for _, lang := range langs {
    beego.Trace("Loading language: " + lang)
    if err := i18n.SetMessage(lang, "conf/"+"locale_"+lang+".ini"); err != nil {
        beego.Error("Fail to set message file: " + err.Error())
        return
    }
}
```

在这段代码中，我们首先从配置文件中获取我们需要支持的语言种类，例如官网支持的语言有 `en-US` 和 `zh-CN` 。接着初始化了一个用于实现用户自由切换语言的 slice（此处不做讨论），最后，根据我们需要支持的语言种类，采用一个循环内调用 `i18n.SetMessage` 加载所有本地化文件。此时，您应该明白为什么我们推荐您采用标准化的形式命名您的本地化文件。

初始化控制器语言

下面的代码摘取自 beego 官网的控制器语言处理部分 `routers/router.go` ，依次根据 URL 指定、Cookies 和浏览器 Accept-Language 来获取用户语言选项，然后设置控制器级别的语言。

```
// setLangVer sets site language version.
func (this *baseRouter) setLangVer() bool {
    isNeedRedir := false
    hasCookie := false

    // 1. Check URL arguments.
    lang := this.Input().Get("lang")

    // 2. Get language information from cookies.
    if len(lang) == 0 {
        lang = this.Ctx.GetCookie("lang")
        hasCookie = true
    } else {
        isNeedRedir = true
    }

    // Check again in case someone modify by purpose.
    if !i18n.IsExist(lang) {
        lang = ""
        isNeedRedir = false
        hasCookie = false
    }

    // 3. Get language information from 'Accept-Language'.
    if len(lang) == 0 {
        al := this.Ctx.Request.Header.Get("Accept-Language")
        if len(al) > 4 {
            al = al[:5] // Only compare first 5 letters.
            if i18n.IsExist(al) {
                lang = al
            }
        }
    }

    // 4. Default language is English.
    if len(lang) == 0 {
        lang = "en-US"
        isNeedRedir = false
    }
```

```
    curLang := langType{
        Lang: lang,
    }

    // Save language information in cookies.
    if !hasCookie {
        this.Ctx.SetCookie("lang", curLang.Lang, 1<<31-1, "/")
    }

    restLangs := make([]*langType, 0, len(langTypes)-1)
    for _, v := range langTypes {
        if lang != v.Lang {
            restLangs = append(restLangs, v)
        } else {
            curLang.Name = v.Name
        }
    }

    // Set language properties.
    this.Lang = lang
    this.Data["Lang"] = curLang.Lang
    this.Data["CurLang"] = curLang.Name
    this.Data["RestLangs"] = restLangs

    return isNeedRedir
}
```

其中， `isNeedRedir` 变量用于表示用户是否是通过 URL 指定来决定语言选项的，为了保持 URL 整洁，官网在遇到这种情况时自动将语言选项设置到 Cookies 中然后重定向。

代码 `this.Data["Lang"] = curLang.Lang` 是将用户语言选项设置到名为 `Lang` 的模板变量中，使得能够在模板中处理语言问题。

以下两行：

```
this.Data["CurLang"] = curLang.Name
this.Data["RestLangs"] = restLangs
```

主要用于实现用户自由切换语言，具体实现原理请参考 beego 官网源码。

控制器语言处理

当作为匿名字段嵌入到 `baseController` 之后，直接通过 `this.Tr(format string, args ...interface{})` 即可进行语言处理。

在模板中使用

通过在控制器中传入一个 `Lang` 变量来指示语言选项后，就可以在模板中进行本地化处理，不过在这之前，需要先注册一个模板函数。

以下代码摘取自 beego 官网源码 `beeweb.go` ：

```
beego.AddFuncMap("i18n", i18n.Tr)
```

注册完成之后，便可配合 `Lang` 变量在模板中进行语言处理：

```
{{i18n .Lang "hi%d" 12}}
```

以上代码会输出：

- 英文 `en-US` ： `hello12`
- 中文 `zh-CN` ： `您好12`

分区功能

针对不同页面，同一个键的名称很可能会对应不同的含义。因此，i18n 模块还利用 INI 格式配置文件的节特性来实现分区功能。

例如，同样是键名 `about` ，在首页需要显示为 `关于` ，而在关于页面需要显示为 `关于我们` ，则可以通过分区功能来实现。

本地化文件中的内容：

```
about = About

[about]
about = About Us
```

获取首页的 `about` ：

```
{{i18n .Lang "about"}}
```

获取关于页面的 `about` ：

```
{{i18n .Lang "about.about"}}
```

歧义处理

由于 `.` 是作为分区的标志，所以当您的键名出现该符号的时候，会出现歧义导致语言处理失败。这时，您只需要在整个键名前加上一个额外的 `.` 即可避免歧义。

例如，我们的键名为 `about.` ，为了避免歧义，我们需要使用：

```
{{i18n .Lang ".about."}}
```

来获取正确的本地化结果。

命令行工具

i18n 模块提供命令行工具 beei18n 来帮助简化开发中的一些步骤。您可以通过以下方式安装：

```
go get github.com/beego/i18n/beei18n
```

同步本地化文件

命令 `sync` 允许您使用已经创建好的一个本地化文件为模板，创建或同步其它的本地化文件：

```
beei18n sync srouce_file.ini other1.ini other2.ini
```

该命令可以同时操作 1 个或多个文件。

其它说明

如果未找到相应键的对应值，则会输出键的原字符串。例如：当键为 `hi` 但未在本地化文件中找到以该字符串命名的键，则会将 `hi` 作为字符串返回给调用者。

beego高级编程

进程内监控

API自动化文档

进程内监控

进程内监控

前面介绍了 toolbox 模块，beego 默认是关闭的，在进程开启的时候监控端口，但是默认是监听在 `127.0.0.1:8088` ，这样无法通过外网访问。当然你可以通过各种方法访问，例如 nginx 代理。

> s="default">
>
> s="default">
>
> 为了安全，建议用户在防火墙中把 8088 端口给屏蔽了。你可以在 conf/app.conf 中打开它

默认监控是关闭的，你可以通过设置参数配置开启监控：

```
EnableAdmin = true
```

而且你还可以修改监听的地址和端口：

```
AdminAddr = "localhost"
AdminPort = 8088
```

打开浏览器，输入 URL： `http://localhost:8088/` ，你会看到一句欢迎词： `Welcome to Admin Dashboard` 。

目前由于刚做出来第一版本，因此还需要后续继续界面的开发。

请求统计信息

访问统计的 URL 地址 `http://localhost:8088/qps` ，展现如下所示：

性能调试

你可以查看程序性能相关的信息，进行性能调优.

健康检查

需要手工注册相应的健康检查逻辑，才能通过 URL `http://localhost:8088/healthcheck` 获取当前执行的健康检查的状态。

定时任务

用户需要在应用中添加了 task，才能执行相应的任务检查和手工触发任务。

- 检查任务状态 URL：`http://localhost:8088/task`
- 手工执行任务 URL：`http://localhost:8088/task?taskname=任务名`

配置信息

应用开发完毕之后，我们可能需要知道在运行的进程到底是怎么样的配置，beego 的监控模块提供了这一功能。

- 显示所有的配置信息: `http://localhost:8088/listconf?command=conf`
- 显示所有的路由配置信息: `http://localhost:8088/listconf?command=router`
- 显示所有的过滤设置信息: `http://localhost:8088/listconf?command=filter`

API自动化文档

API自动化文档

自动化文档一直是我梦想中的一个功能，这次借着公司的项目终于实现了出来，我说过 beego 不仅仅要让开发 API 快，而且让使用 API 的用户也能快速的使用我们开发的 API，这个就是我开发这个项目的初衷。好了，赶紧动手实践一把吧，首先 `bee api beeapi` 新建一个 API 应用做起来吧。

API 全局设置

必须设置在 `routers/router.go` 中，文件的注释，最顶部：

```
// @APIVersion 1.0.0
// @Title mobile API
// @Description mobile has every tool to get any job done, so codename for the new mobile APIs.
// @Contact astaxie@gmail.com
package routers
```

全局的注释如上所示，是显示给全局应用的设置信息，有如下这些设置

- @APIVersion
- @Title
- @Description
- @Contact
- @TermsOfServiceUrl
- @License
- @LicenseUrl

路由解析须知

目前自动化文档只支持如下的写法的解析，其他写法函数不会自动解析，即 namespace+Include 的写法，而且只支持二级解析，一级版本号，二级分别表示应用模块

```
func init() {
    ns :=
        beego.NewNamespace("/v1",
            beego.NSNamespace("/customer",
                beego.NSInclude(
                    &controllers.CustomerController{},
                    &controllers.CustomerCookieCheckerController{},
                ),
            ),
            beego.NSNamespace("/catalog",
                beego.NSInclude(
                    &controllers.CatalogController{},
                ),
            ),
            beego.NSNamespace("/newsletter",
                beego.NSInclude(
                    &controllers.NewsLetterController{},
                ),
            ),
            beego.NSNamespace("/cms",
                beego.NSInclude(
                    &controllers.CMSController{},
                ),
            ),
            beego.NSNamespace("/suggest",
                beego.NSInclude(
                    &controllers.SearchController{},
                ),
            ),
        )
    beego.AddNamespace(ns)
}
```

应用注释

接下来就是我们最重要的注释了，就是我们定义的，我们先来看一个例子：

```
package controllers

import "github.com/astaxie/beego"

// CMS API
type CMSController struct {
    beego.Controller
```

```
}

func (c *CMSController) URLMapping() {
    c.Mapping("StaticBlock", c.StaticBlock)
    c.Mapping("Product", c.Product)
}

// @Title getStaticBlock
// @Description get all the staticblock by key
// @Param    key         path     string    true         "The email for login"
// @Success 200 {object} models.ZDTCustomer.Customer
// @Failure 400 Invalid email supplied
// @Failure 404 User not found
// @router /staticblock/:key [get]
func (c *CMSController) StaticBlock() {

}

// @Title Get Product list
// @Description Get Product list by some info
// @Success 200 {object} models.ZDTProduct.ProductList
// @Param    category_id        query    int    false        "category id"
// @Param    brand_id    query    int    false        "brand id"
// @Param    query    query    string    false        "query of search"
// @Param    segment    query    string    false        "segment"
// @Param    sort    query    string    false        "sort option"
// @Param    dir    query    string    false        "direction asc or desc"
// @Param    offset    query    int    false        "offset"
// @Param    limit    query    int    false        "count limit"
// @Param    price            query    float        false        "price"
// @Param    special_price    query    bool        false        "whether this i
s special price"
// @Param    size             query    string        false        "size filter"
// @Param    color            query    string        false        "color filte
r"
// @Param    format             query    bool        false        "choose return
format"
// @Failure 400 no enough input
// @Failure 500 get products common error
// @router /products [get]
func (c *CMSController) Product() {

}
```

首先是 CMSController 定义上面的注释，这个是用来显示这个模块的作用。接下来就是每一个函数上面的注释，这里列出来支持的各种注释：

- @Title

  这个 API 所表达的含义，是一个文本，空格之后的内容全部解析为 title

- @Description

  这个 API 详细的描述，是一个文本，空格之后的内容全部解析为 Description

- @Param

  参数，表示需要传递到服务器端的参数，有五列参数，使用空格或者 tab 分割，五个分别表示的含义如下

  i. 参数名
  ii. 参数类型，可以有的值是 formData、query、path、body、header，formData 表示是 post 请求的数据，query 表示带在 url 之后的参数，path 表示请求路径上得参数，例如上面例子里面的 key，body 表示是一个 raw 数据请求，header 表示带在 header 信息中得参数。
  iii. 参数类型
  iv. 是否必须
  v. 注释

- @Success

  成功返回给客户端的信息，三个参数，第一个是 status code。第二个参数是返回的类型，必须使用 {} 包含，第三个是返回的对象或者字符串信息，如果是 {object} 类型，那么 bee 工具在生成 docs 的时候会扫描对应的对象，这里填写的是想对你项目的目录名和对象，例如 `models.ZDTProduct.ProductList` 就表示 `/models/ZDTProduct` 目录下的 `ProductList` 对象。

  ```
  s="default">

  s="default">

  三个参数必须通过空格分隔
  ```

- @Failure

失败返回的信息，包含两个参数，使用空格分隔，第一个表示 status code，第二个表示错误信息

- @router

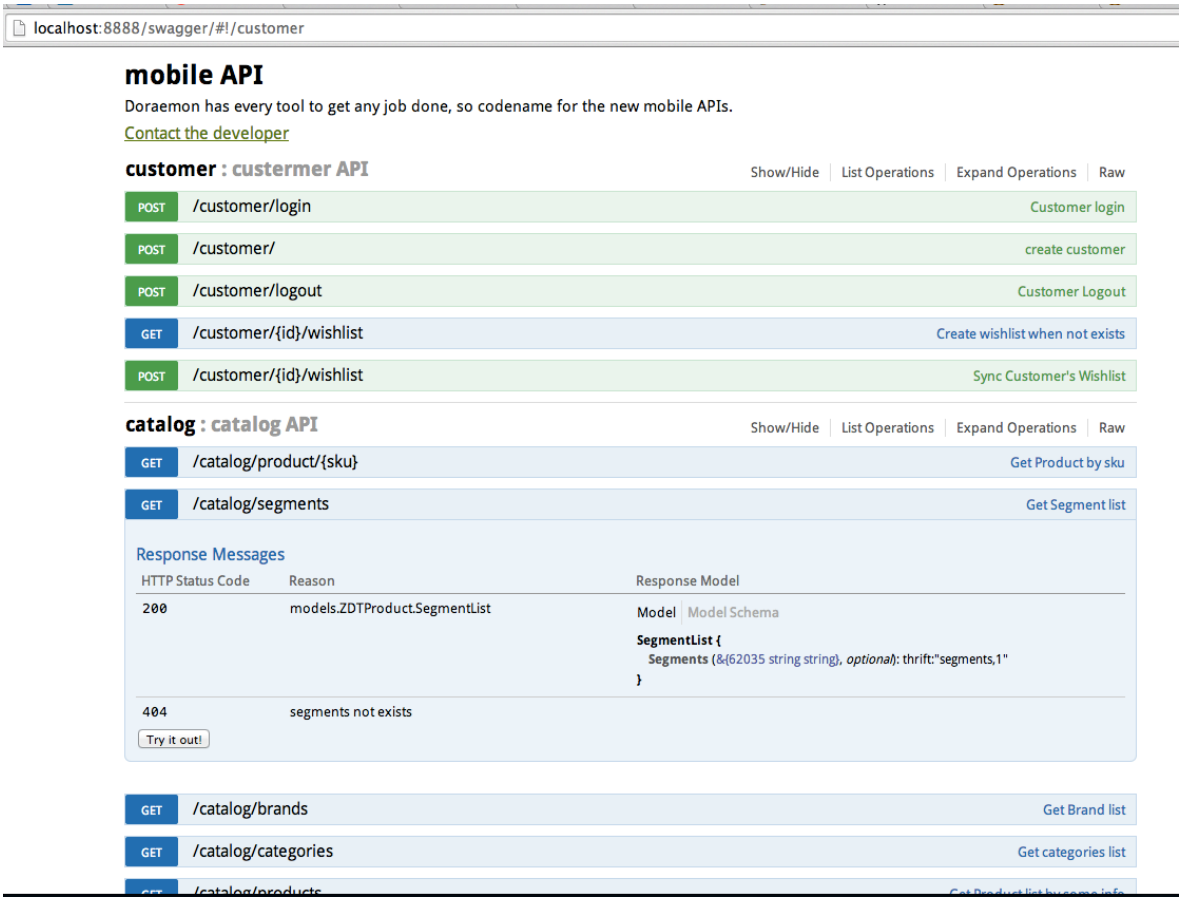
  路由信息，包含两个参数，使用空格分隔，第一个是请求的路由地址，支持正则和自定义路由，和之前的路由规则一样，第二个参数是支持的请求方法,放在 `[]` 之中，如果有多个方法，那么使用 `,` 分隔。

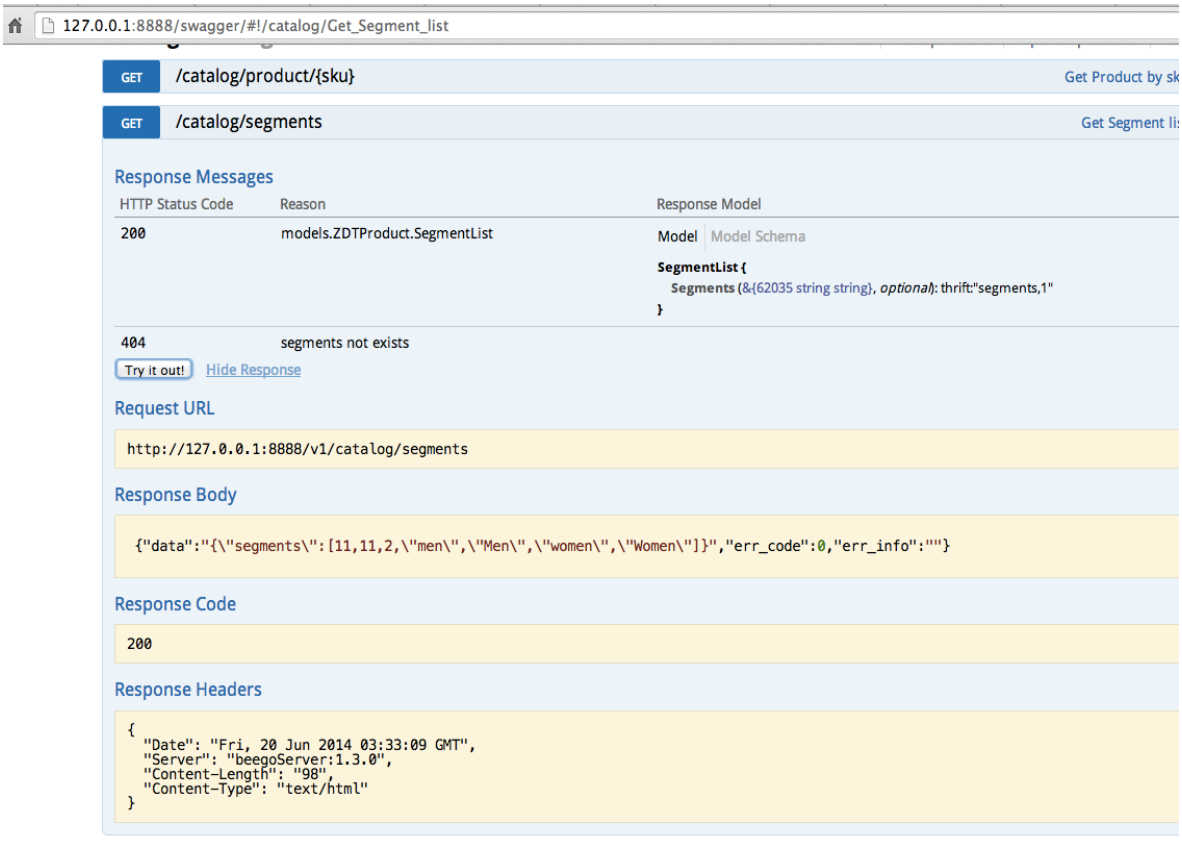
如何自动化生成文档

要使得文档工作，你需要做几个事情，

- 第一开启应用内文档开关，在配置文件中设置：`EnableDocs = true`,
- 然后在你的 `main.go` 函数中引入 `_ "beeapi/docs"`（beego 1.7.0 之后版本不需要添加该引用）。
- 这样你就已经内置了 docs 在你的 API 应用中，然后你就使用 `bee run -gendoc=true -downdoc=true`,让我们的 API 应用跑起来，`-gendoc=true` 表示每次自动化的 build 文档，`-downdoc=true` 就会自动的下载 swagger 文档查看器

好了，现在打开你的浏览器查看一下效果，是不是已经完美了。下面是我的 API 文档效果：


localhost:8888/swagger/#!/customer
mobile API
Doraemon has every tool to get any job done, so codename for the new mobile APIs.
Contact the developer
customer : custermer API
Show/Hide | List Operations | Expand Operations | Raw
POST /customer/login Customer login
POST /customer/ create customer
POST /customer/logout Customer Logout
GET /customer/{id}/wishlist Create wishlist when not exists
POST /customer/{id}/wishlist Sync Customer's Wishlist
catalog : catalog API
Show/Hide | List Operations | Expand Operations | Raw
GET /catalog/product/{sku} Get Product by sku
GET /catalog/segments Get Segment list
Response Messages
HTTP Status Code
Reason
Response Model
200
models.ZDTProduct.SegmentList
Model | Model Schema
SegmentList {
Segments (&{62035 string string}, optional): thrift:"segments,1"
}
404
segments not exists
Try it out!
GET /catalog/brands Get Brand list
GET /catalog/categories Get categories list


127.0.0.1:8888/swagger/#!/catalog/Get_Segment_list
GET /catalog/product/{sku} Get Product by sk
GET /catalog/segments Get Segment li
Response Messages
HTTP Status Code
Reason
Response Model
200
models.ZDTProduct.SegmentList
Model | Model Schema
SegmentList {
Segments (&{62035 string string}, optional): thrift:"segments,1"
}
404
segments not exists
Try it out!
Hide Response
Request URL
http://127.0.0.1:8888/v1/catalog/segments
Response Body
{"data":"{\"segments\":[11,11,2,\"men\",\"Men\",\"women\",\"Women\"]}","err_code":0,"err_info":""}
Response Code
200
Response Headers
{
"Date": "Fri, 20 Jun 2014 03:33:09 GMT",
"Server": "beegoServer:1.3.0",
"Content-Length": "98",
"Content-Type": "text/html"
}

可能遇到的问题

1. CORS
   两种解决方案：

   - 把 swagger 集成到应用中，下载请到swagger,然后放在项目目录下：

   ```
   if beego.BConfig.RunMode == "dev" {
       beego.BConfig.WebConfig.DirectoryIndex = true
       beego.BConfig.WebConfig.StaticDir["/swagger"] = "swagger"
   }
   ```

   - API 增加 CORS 支持

   ```
   ctx.Output.Header("Access-Control-Allow-Origin", "*")
   ```

2. 未知错误，因为这是我自己项目中使用的，所以可能大家在写的过程中会遇到一些莫名的错误，请提 issue 去吧！

应用部署

独立部署

Supervisor部署

Nginx部署

Apache部署

独立部署

独立部署

独立部署即为在后端运行程序，让程序跑在后台。

linux

在 linux 下面部署，我们可以利用 `nohup` 命令，把应用部署在后端，如下所示：

```
nohup ./beepkg &
```

这样你的应用就跑在了 Linux 系统的守护进程

Windows

在 Windows 系统中，设置开机自动，后台运行，有如下几种方式：

1. 制作 bat 文件，放在“启动”里面
2. 制作成服务

Supervisor部署

Supervisor部署

Supervisord 是用 Python 实现的一款非常实用的进程管理工具，supervisord 还要求管理的程序是非 daemon 程序，supervisord 会帮你把它转成 daemon 程序，因此如果用 supervisord 来管理 nginx 的话，必须在 nginx 的配置文件里添加一行设置 daemon off 让 nginx 以非 daemon 方式启动。

supervisord 安装

1. 安装 setuptools

```
wget https://pypi.python.org/packages/2.7/s/setuptools/setuptools-0.6c11-py2.7.egg

sh setuptools-0.6c11-py2.7.egg

easy_install supervisor

echo_supervisord_conf >/etc/supervisord.conf

mkdir /etc/supervisord.conf.d
```

2. 修改配置 `/etc/supervisord.conf`

```
[include]
files = /etc/supervisord.conf.d/*.conf
```

3. 新建管理的应用

```
cd /etc/supervisord.conf.d
vim beepkg.conf
```

配置文件：

```
[program:beepkg]
directory = /opt/app/beepkg
command = /opt/app/beepkg/beepkg
```

```
autostart = true
startsecs = 5
user = root
redirect_stderr = true
stdout_logfile = /var/log/supervisord/beepkg.log
```

supervisord 管理

Supervisord 安装完成后有两个可用的命令行 supervisord 和 supervisorctl，命令使用解释如下：

- supervisord，初始启动 Supervisord，启动、管理配置中设置的进程。
- supervisorctl stop programxxx，停止某一个进程(programxxx)，programxxx 为 [program:beepkg] 里配置的值，这个示例就是 beepkg。
- supervisorctl start programxxx，启动某个进程
- supervisorctl restart programxxx，重启某个进程
- supervisorctl stop groupworker: ，重启所有属于名为 groupworker 这个分组的进程 (start,restart 同理)
- supervisorctl stop all，停止全部进程，注：start、restart、stop 都不会载入最新的配置文件。
- supervisorctl reload，载入最新的配置文件，停止原有进程并按新的配置启动、管理所有进程。
- supervisorctl update，根据最新的配置文件，启动新配置或有改动的进程，配置没有改动的进程不会受影响而重启。

> s="default">
>
> s="default">
>
> 注意：显示用 stop 停止掉的进程，用 reload 或者 update 都不会自动重启。

Nginx部署

Nginx部署

Go 是一个独立的 HTTP 服务器，但是我们有些时候为了 nginx 可以帮我做很多工作，例如访问日志，cc 攻击，静态服务等，nginx 已经做的很成熟了，Go 只要专注于业务逻辑和功能就好，所以通过 nginx 配置代理就可以实现多应用同时部署，如下就是典型的两个应用共享 80 端口，通过不同的域名访问，反向代理到不同的应用。

```
server {
    listen       80;
    server_name  .a.com;

    charset utf-8;
    access_log  /home/a.com.access.log;

    location /(css|js|fonts|img)/ {
        access_log off;
        expires 1d;

        root "/path/to/app_a/static";
        try_files $uri @backend;
    }

    location / {
        try_files /_not_exists_ @backend;
    }

    location @backend {
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header Host            $http_host;

        proxy_pass http://127.0.0.1:8080;
    }
}

server {
    listen       80;
    server_name  .b.com;

    charset utf-8;
    access_log  /home/b.com.access.log  main;
```

```
    location /(css|js|fonts|img)/ {
        access_log off;
        expires 1d;

        root "/path/to/app_b/static";
        try_files $uri @backend;
    }

    location / {
        try_files /_not_exists_ @backend;
    }

    location @backend {
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header Host            $http_host;

        proxy_pass http://127.0.0.1:8081;
    }
}
```

Apache部署

Apache部署

Apache 配置

apache 和 nginx 的实现原理一样，都是做一个反向代理，把请求向后端传递，配置如下所示：

```
NameVirtualHost *:80
<VirtualHost *:80>
    ServerAdmin webmaster@dummy-host.example.com
    ServerName www.a.com
    ProxyRequests Off
    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>
    ProxyPass / http://127.0.0.1:8080/
    ProxyPassReverse / http://127.0.0.1:8080/
</VirtualHost>

<VirtualHost *:80>
    ServerAdmin webmaster@dummy-host.example.com
    ServerName www.b.com
    ProxyRequests Off
    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>
    ProxyPass / http://127.0.0.1:8081/
    ProxyPassReverse / http://127.0.0.1:8081/
</VirtualHost>
```

beego第三方库

beego第三方库

随着 beego 的发展, 基于 beego 的第三方库也逐渐的增加,如果大家有基于 beego 的库,欢迎递交你的地址

- gorelic
- 支付宝 SDK
- pongo2
- keenio
- casbin - RBAC ACL plugins

应用例子

在线聊天室

短域名服务

Todo列表

在线聊天室

在线聊天室

本示例通过两种形式展示了如何实现一个在线聊天室应用：

- 使用长轮询模式。
- 使用 WebSocket 模式。

以上两种模式均默认将数据存储在内存中，因此每次启动都会被重置。但您也可以通过修改 `conf/app.conf` 中的设置来启用数据库。

以下为项目组织大纲：

```
WebIM/
    WebIM.go              # main 包的文件
    conf
        app.conf          # 配置文件
    controllers
        app.go              # 供用户选择技术和用户名的欢迎页面
        chatroom.go          # 数据管理相关的函数
        longpolling.go     # 长轮询模式的控制器和方法
        websocket.go     # WebSocket 模式的控制器和方法
    models
        archive.go          # 操作数据相关的函数
    views
        ...                  # 模板文件
    static
        ...                  # JavaScript 和 CSS 文件
```

到 GitHub 上浏览代码

短域名服务

短域名服务

这个例子演示了如何使用 beego 开发 API 应用. 他包含了两个 API 接口:

- /v1/shorten
- /v1/expand

到 GitHub 上浏览代码

Todo列表

Todo列表

An AngularJS + beego project

```
 bee new todo
[INFO] Creating application...
/Users/astaxie/gopath/src/todo/
/Users/astaxie/gopath/src/todo/conf/
/Users/astaxie/gopath/src/todo/controllers/
/Users/astaxie/gopath/src/todo/models/
/Users/astaxie/gopath/src/todo/static/
/Users/astaxie/gopath/src/todo/static/js/
/Users/astaxie/gopath/src/todo/static/css/
/Users/astaxie/gopath/src/todo/static/img/
/Users/astaxie/gopath/src/todo/views/
/Users/astaxie/gopath/src/todo/conf/app.conf
/Users/astaxie/gopath/src/todo/controllers/default.go
/Users/astaxie/gopath/src/todo/views/index.tpl
/Users/astaxie/gopath/src/todo/main.go
13-12-14 10:05:44 [SUCC] New application successfully created!
```

到 GitHub 上浏览代码

beego实用库

验证码

分页

验证码

验证码

分页

分页

提供一个例子

结构

```
type Page struct {
    PageNo      int
    PageSize    int
    TotalPage   int
    TotalCount int
    FirstPage   bool
    LastPage    bool
    List        interface{}
}
func PageUtil(count int, pageNo int, pageSize int, list interface{}) Page {
    tp := count / pageSize
    if count % pageSize > 0 {
        tp = count / pageSize + 1
    }
    return Page{PageNo: pageNo, PageSize: pageSize, TotalPage: tp, TotalCount: c
ount, FirstPage: pageNo == 1, LastPage: pageNo == tp, List: list}
}
```

页面是使用 js 插件进行分页 https://github.com/lyonlai/bootstrap-paginator

```
<script type="text/javascript" src="/static/js/bootstrap-paginator.min.js"></scr
ipt>
<script type="text/javascript">
  $(function () {
    $("#page").bootstrapPaginator({
      currentPage: '{{.Page.PageNo}}',
      totalPages: '{{.Page.TotalPage}}',
      bootstrapMajorVersion: 3,
      size: "small",
      onPageClicked: function(e,originalEvent,type,page){
        window.location.href = "/?p=" + page
      }
    });
```

```
  });
</script>
```

FAQ

FAQ

1. 找不到模板文件，找不到配置文件，nil 指针错误

   这种大多数情况是由于你采用了 `go run main.go` 这样的方式来运行你的应用，go run 是把文件编译之后放在了 tmp 下去运行，而 beego 的应用会读取应用的当前运行目录对应的 conf,view 去查找相应的配置文件和模板，因此要正确运行，请使用 `go build` 然后执行应用 `./app` 这种方式来运行。或者使用 `bee run app` 来启动你的应用。

2. beego 可以应用于生产环境吗？

   目前 beego 已经被广大用户应用于各大生产环境，例如盛大的 CDN 系统，360 的搜索 API、Bmob 移动云 API，weico 的后端 API 应用，还有很多其他 Web 应用和服务器应用，都是一些高并发高性能的应用，所以请放心大胆的使用。

3. beego 将来升级会影响现有的 API 吗？

   beego 从 0.1 版本到现在基本保持了稳定的 API，很多应用都是可以无缝的升级到最新版本的 beego。将来升级重构都会尽量保持 beego 的 API 的稳定性。

4. beego 会持续开发吗？

   很多人使用开源软件都有一个担心就是怕项目不在持续，目前我们 beego 开发组有四个人一直在贡献代码，我想我们能够坚持把这个项目做好，而且会持续不断的进行改进。